Escuela Politécnica Superior

Videojuego multigénero en Unreal Engine 4

Grado en Ingeniería Multimedia

Trabajo Fin de Grado

Autor:

Ángel David González Cobo

Tutor/es:

Carlos José Villagrá Arnedo

Universitat d'Alacant
Universidad de Alicante

Enero 2019

A toda la gente que me ha apoyado estos años de aprendizaje, sobre todo a los amigos, compañeros que hemos estado haciendo equipo en los buenos y malos momentos durante toda esta etapa. También a mis seres queridos que me han apoyado durante estos años.

Índices

Índice de contenidos

Índice de figuras

Índice de tablas

1. Introducción

La industria del videojuego es una industria al alza que actualmente da empleo a una gran cantidad de personas dedicadas a diferentes ámbitos dentro de los estudios de los videojuegos. En lo referente a generación de ingresos y de beneficios, solo en España, el mercado de los videojuegos ya factura más del doble de los sectores de cine y música juntos.

Debido a este crecimiento y popularidad, hace que estén surgiendo actualmente estudios con un número reducido de personas o incluso en solitario, dedicándose a desarrollar videojuegos de manera independiente de las distribuidoras de alto presupuesto. Estos estudios y videojuegos son los llamados *indies*.

El objetivo de este proyecto es la simulación de desarrollo de un proyecto como un estudio *indie*, desde el comienzo a su finalización, partiendo de una base de programación, pero sin conocimiento del entorno de desarrollo. Se analizarán, definirán y desarrollarán diferentes mecánicas y elementos gráficos dentro de un mercado emergente.

Este documento está separado en diversos elementos para facilitar la introducción y lectura de los elementos desarrollados en el proyecto realizado.

Por orden de aparición, primero se encuentra el *Marco teórico o Estado de arte* donde se detalla el concepto de videojuego, los géneros de videojuegos que existen y los referentes en los que se ha basado el proyecto. Posteriormente se encuentran los *Objetivos* que se quieren conseguir con el proyecto. En el siguiente apartado, *Metodología*, se comentan tanto las herramientas utilizadas, como la metodología de desarrollo utilizada.

En el apartado del *Cuerpo del trabajo* se encuentra la parte principal de este documento, donde se analizan las herramientas que existen en el mercado para el desarrollo de videojuegos y el documento de diseño de videojuego, donde se define el concepto, géneros a implementar, estilo visual, mecánicas, etc. Además, se encuentra dentro de esta sección un apartado importante, el *Desarrollo e implementación donde* se detallan en mayor profundidad los elementos creados durante el proceso de desarrollo del videojuego y el estado final de los mismos.

Por último, se encuentra la sección de *Conclusiones* donde se comentan diferentes aspectos del proyecto durante su desarrollo, así como el grado de consecución de los objetivos planteados al inicio de este proyecto y, además, un comentario respecto del registro de tiempos realizado durante todo el proceso.

2. Marco teórico o Estado del arte

En este apartado se explicará el concepto de videojuego, así como los referentes en los que se han fijado diferentes funcionalidades, mecánicas y estilos visuales de diversos videojuegos, para el desarrollo del proyecto.

Para desarrollar un videojuego (sobre todo en las primeras fases) una de las mejores técnicas para imaginar o definir lo que queremos conseguir es buscar referentes en otros videojuegos. Esto hace que a la hora de definir las mecánicas que queremos que posean los personajes dentro del juego, o los elementos que aparezcan dentro del mismo se puedan imaginar de manera inmediata. Este sistema se ha utilizado en el desarrollo de este proyecto. Entre los videojuegos referentes nos encontraremos tanto producciones de alto presupuesto llamados comúnmente *Triple A*, producidos, y distribuidos por empresas de gran presupuesto, como videojuegos desarrollados por empresas *indies* de bajo presupuesto, en las que la misma empresa que lo desarrolla se encargaría de distribuirlo en el mercado.

2.1. Concepto de videojuego

Un videojuego es "*Juego electrónico que se visualiza en una pantalla.*" (Real Academia Española, 2019). El objetivo básico de un videojuego está basado en el entretenimiento del usuario, pero se pueden dar diversos escenarios en los que el videojuego tenga como objetivo secundario diversos elementos, como pueda ser ejercitar al usuario, hacer de rehabilitación de ciertas lesiones, aprendizaje de diferentes campos, etc.

Durante el paso de los años se han ido perfeccionando técnicas y herramientas para el desarrollo de estos videojuegos, creando una industria a nivel global que, en lo referente al

consumo, hace frente a industrias como la del cine. Hay gran cantidad de videojuegos y de diferentes clases, por ello, se dividen en géneros para poder clasificarlos y distinguirlos.

2.1.1. Géneros de videojuegos

Los géneros de videojuegos son maneras de clasificarlos. A diferencia de otros sectores como pueda ser la literatura o el cine, que se clasifican dependiendo de la historia o la ambientación de la obra, en este caso, los videojuegos son clasificados dependiendo de las mecánicas que posean. En este caso nos encontramos con un juego que implementa géneros diferentes, que son *Hack 'n' Slash* o *Hack and Slash*, *RPG* con combate por turnos y sigilo.

El género *Hack 'n' Slash* centra sus mecánicas en el combate, en vez de la narración y la historia. Este género entraría dentro de un género global más amplio que sería el género de acción. Los juegos de este género poseen un sistema de combate en tiempo real comúnmente con un sistema de combos para realizar los ataques de los personajes.

En el lado contrario al *Hack 'n' Slash*, tenemos el género de sigilo. Este género centra las mecánicas de juego o *gameplay*, en evitar la confrontación con los antagonistas del videojuego. Una de las mecánicas que suelen tener estos géneros, es una habilidad del personaje para eliminar un enemigo que haya detectado al jugador.

Estos dos géneros están siendo combinados en la actualidad en gran parte de los videojuegos, ya que dejar la libertad de escoger el camino a seguir al jugador. Dejan un camino centrado en el combate con los enemigos, y otro camino escabulléndose de ellos.

Por último, tendríamos el género de *RPG (Role-playing game* o juego de rol*)* con sistema de combate por turnos. Los *RPG* son videojuegos que nacieron a partir de los juegos de rol de mesa y suelen compartir terminología, o mecánicas de estos juegos. La variante de estos juegos, en los que entraría los *RPG* con sistema de combate por turnos serían los *RPG* tácticos. Estos videojuegos basan el combate en un gestor de turnos donde de manera aleatoria o basándolo alguna estadística de los participantes asigna un orden de turnos. Para entrar en estos combates, comúnmente suelen haber dos variantes. Los videojuegos donde el jugador maneja un personaje por un mundo abierto, y al caminar o colisionar con ciertos

elementos entran en la escena de combate o, por otro lado, el jugador navega a través de diferentes diálogos o niveles, seleccionando opciones y entrando en dichos combates.

2.2. Videojuegos referentes

En esta sección se mostrarán los distintos juegos que se han tomado como referencia para desarrollar este videojuego.

Los elementos que se buscan a la hora de elegir los referentes han sido en relación con las mecánicas deseadas en los diferentes elementos. Por ejemplo, los referentes para hacer el género *Hack 'n' Slash* son juegos en los que el combate es en tiempo real, se pulsan diferentes botones y se accionan habilidades en el combate para resolverlos. Para el combate por turnos se han elegido los juegos más famosos y mejor valorados personalmente en el género que se desea conseguir.

2.2.1. Zelda Breath of the Wild

Zelda Breath of the Wild, desarrollado por *Nintendo*, en colaboración con *Monolith Soft*. Es un videojuego de acción y aventuras basado en la libertad de acción del usuario. Desde la primera presentación que hizo *Nintendo* varios años antes del lanzamiento, el juego ha sido esperado positivamente por el público. Una vez fue lanzado en 2017 para *Nintendo Switch* y *Wii U*, recibió 3 premios en *The Game Awards 2017*, mejor dirección, mejor juego de acción/aventuras y por último mejor juego del año.

Ilustración 1. Escena de Zelda Breath of the Wild. Fuente: https://www.3djuegos.com/25275/the-legend-of-zelda-breath-of-the-wild

Este videojuego se ha seleccionado en parte por las mecánicas que posee de combate en tiempo real y por el estilo visual tan característico que posee el videojuego. Este es uno de los videojuegos referente en cuanto se menciona el estilo de iluminación y sombreado de *Cel Shading*. Como se puede observar en la ***ilustración 1*** el estilo visual del videojuego combine texturas muy simples y colores planos, con elementos visuales parece que se han pintado con pincel. Esto hace que se combine el sombreado e iluminación del estilo *Cel Shading* como son las sombras marcadas con escasa difuminación, con elementos que han sido texturizados de manera detallada y simple a la vez, consiguiendo una ambientación perfecta para este videojuego.

2.2.2. Saga Darksiders

La saga de videojuegos de *Darksiders*, videojuegos de acción, aventuras con toques de RPG, desarrollado por *Vigil Games* y producido por *THQ*. El primer juego lanzado en 2010 para las plataformas *Xbox 360 y PlayStation 3.*

Durante el desarrollo de la historia del videojuego, el jugador va desbloqueando elementos y equipamiento para el personaje que se maneja en el videojuego. Estos videojuegos combinan combates centralizados en realizar una serie de combos con las armas que se

disponen y eventos cinemáticos que se accionan como se puede ver en la ***ilustración 2*** para terminar con los enemigos de una manera espectacular.

Ilustración 2. Escena de combate en Darksiders. Fuente: https://www.3djuegos.com/juegos/imagenes/8484/0/darksiders/

Durante la partida, el jugador se sitúa en un mundo postapocalíptico en el que el jugador maneja a un jinete del apocalipsis, siendo Guerra en la primera edición y utiliza una gran espada. En la segunda edición de los videojuegos, el jugador maneja el jinete Muerte, que blande una gran guadaña que divide el personaje en dos pequeñas para realizar ataques más rápidos. Por último, en la tercera edición de la saga manejamos a Furia, que representa a Hambre en los jinetes tradicionales. Este personaje blande un látigo de fuego.

2.2.3. Dishonored

Dishonored es un videojuego de acción, aventura y sigilo en primera persona desarrollado por *Arkane Studios* y publicado por *Bethesda Softworks en 2012*. El videojuego esta desarrollado en *Unreal Engine 3*, una versión anterior al motor gráfico que se ha utilizado en este proyecto. Este videojuego tuvo muy buena acogida entre el público y recibió varios premios, como fue "*Mejor juego de acción*" en *IGN's Best Of 2012 awards* y recibió el premio a "*Mejor juego*" en los premios *British Academy Games Awards.*

Durante el desarrollo de los niveles y misiones el jugador tiene la elección de decidir como realizará el objetivo actual, ya sea, de manera sigilosa entrando por laterales o tejados de las casas, o ya sea entrando y eliminando todos los enemigos que se encuentren a su paso.

La historia de este videojuego transcurre en una ciudad llamada *Dunwall* con una ambientación inspirado en Londres en el final del siglo XIX. El jugador encarna a *Corvo*, el guardaespaldas de la emperatriz, donde este se ve envuelto en una conspiración sobre su persona, al ser acusado de asesinato de la emperatriz. Durante el desarrollo de la historia el personaje tendrá que actuar de asesino para desenmascarar a los culpables de la conspiración.

Ilustración 3. Escena de juego de Dishonored. Fuente: https://www.3djuegos.com/11694/dishonored/

2.2.4. Saga Final Fantasy

La saga *Final Fantasy* es una saga de videojuegos principalmente del género *RPG* desarrollados y publicados por la compañía *Squaresoft*. La serie de videojuegos se inició en 1987. Su creador, *Hironobu Sakaguchi*, utilizó sus últimos recursos en realizar esta producción, y como conclusión a su carrera en caso de que no hubiese tenido éxito. Debido al éxito que tuvo y siguen teniendo estos videojuegos, la saga *Final Fantasy* cuenta con

gran cantidad de tanto de ediciones y spin off de diferentes géneros, así como de premios y galardones en diferentes categorías para sus videojuegos.

Para los combates por turnos este es uno de los más grandes y destacados juegos que se podrían tomar como referentes como puede verse en la ***ilustración 4***. La saga *Final Fantasy* lleva muchos videojuegos en su trayectoria, y la gran mayoría son en este es estilo de combate por turnos para el desarrollo de los personajes que controla el jugador. Aunque el sistema de combate y desarrollo de personaje es un gran atractivo para los jugadores, el punto fuerte de sus ediciones está en la historia en la que se ven envueltos los personajes de sus videojuegos, haciendo así, que el jugador quede inmerso en la historia desde el inicio de la partida.

Ilustración 4. Escena de combate en World of Final Fantasy. Fuente: https://www.3djuegos.com/juegos/imagenes/22685/0/world-of-final-fantasy

2.2.5. Saga Pokémon

La saga *Pokémon* es una saga de videojuegos *RPG* desarrollados por *Game Freak* y publicados por la compañía *Nintendo*. Fue lanzado el primer videojuego en Asia en 1996 y no llegaría a Europa hasta finales de 1999. Las primeras ediciones de estos juegos fueron lanzados para la consola portátil de *Nintendo*, la *Game Boy*.

En los inicios de esta saga, gráficamente tenían grandes limitaciones como se puede observan en la ***ilustración 5***. Aun así, estos juegos implementaban 151 *Pokémon* que se podían capturar, un gran número de ataques que aprendían.

Ilustración 5. Escena de combate en Pokémon Rojo/Azul. Fuente: https://fi.wikipedia.org/wiki/Pok%C3%A9mon_Red_ja_Blue

Los juegos de *Pokémon* están situados en diferentes regiones ficticias. Las primeras ediciones estaban situadas en *Kanto*, donde el jugador controlaba a un personaje que quería convertirse en el mejor entrenador. Para conseguir este fin, debería explorar la región de *Kanto*, y debía ir derrotando los líderes de los gimnasios para obtener las medallas de los líderes. Una vez obtenidas debería derrotar al alto mando de la liga *Pokémon*. Una vez conseguido este logro, le quedaría capturar a los restantes *Pokémon* que le quedasen para llegar a obtener los 151 *Pokémon* existentes en esas ediciones.

Estas ediciones marcaron el inicio de una gran saga de juegos, con la que actualmente se encuentran más de 30 ediciones y *spin off* de la saga. Es por esto por lo que en 2009 fueron incluidos en el *Libro Guinness de los récords mundiales* en las secciones de "*Mejor juego vendido del género RPG de todos los tiempos*" y "*Mejor juego de rol vendido para Game Boy*".

Ilustración 6. Escena de combate de Pokémon Let's Go Pikachu. Fuente: https://www.3djuegos.com/juegos/imagenes/29790/0/pokemon-lets-go-pikachu-pokemon-lets-go-eevee

2.2.6. Aragami

Aragami es un videojuego creado por la compañía española *Lince Works* y publicado por la misma compañía en 2016 para las plataformas *Linux*, *Microsoft Windows* y *PlayStation 4*. El videojuego está desarrollado en el motor *Unity3D* y entraría dentro de la descripción de un videojuego *indie.*

Ambientado en un medievo asiático, el personaje controla un espíritu de sombra (*Aragami*) para realizar los distintos objetivos del mapa y eliminar a los enemigos que tienen el poder de controlar la luz. Durante el desarrollo de la historia, el personaje puede moverse y teletransportarse a través de los puntos de sombras del mapa para evitar ser descubierto, y puede eliminar los enemigos sin ser descubierto como se ve en la ***ilustración 7***. A medida que avance la historia, el jugador mejora las habilidades del personaje al obtener unos pergaminos que están esparcidos por los diferentes mapas.

Ilustración 7. Escena de asesinato en sigilo en Aragami. Fuente: https://www.3djuegos.com/juegos/imagenes/25180/0/aragami/

3. Objetivos

Para la realización del proyecto se han planteado una serie de diferentes objetivos. Como objetivo principal y el más importante dentro de todos los objetivos planteados sería el objetivo de desarrollar un videojuego multigénero dentro de la plataforma que proporciona *Unreal Engine*.

Se partirá sin ninguna base de conocimiento especifica del entorno, lo que conlleva que hay que aprender a desarrollar en el entorno de *Unreal Engine*. Se parte con una base de programación en *C++*, de manera que los aspectos generales de la programación son conocidos, pero los elementos específicos dentro del entornos son completamente desconocidos. Además, ya que dentro de *Unreal Engine 4* hay diferentes maneras de crear los diferentes elementos, que son mediante código tradicional en *C++*, o mediante una programación visual basada en nodos llamados *Blueprints*, se aprovechara para desarrollar en ambos sentidos, para comprobar la eficacia y comodidad de los diferentes entornos de programación.

El videojuego que se va a desarrollar contará con diferentes géneros: *Hack 'n' Slash*, sigilo y combate por turnos. Dependiendo del nivel en el que se encuentre el jugador será un género distinto. Para el desarrollo de elementos gráficos se crearán parte de ellos y además se integrarán diferentes paquetes de elementos creados para aprender sobre la importación de diferentes elementos externos en un proyecto.

Para mejorar la compresión de los puntos comentados anteriormente se han acotado los elementos en formato de lista:

- Desarrollar un videojuego multigénero en *Unreal Engine 4*.
- Aprendizaje de *Unreal Engine 4* con *Blueprints* y *C++*.
- Combinación de desarrollo entre ambas opciones: *Blueprints* y *C++*.
- Implementación y gestión de múltiples géneros en un mismo proyecto.
- Desarrollo y gestión de un videojuego en todos los aspectos de manera individual.
- Importación y utilización de paquetes de recursos en el proyecto.
- Seguimiento del desarrollo del producto desde el inicio hasta el final.

4. Metodología

Para el desarrollo del proyecto, se ha hecho un seguimiento de las tareas y de los tiempos invertidos en cada una de estas y en los cambios que se han ido realizando durante el desarrollo. El desarrollo del proyecto ha sido con una metodología basada en prototipos de forma incremental, es por esto por lo que es necesario un control de versiones para tener un seguimiento del desarrollo del proyecto. Para empezar, se han desarrollado diferentes mecánicas de manera aislada para familiarizarse con el entorno de trabajo.

Una vez terminadas las diferentes mecánicas de manera aislada se ha creado un proyecto donde se irán incluyendo estas mecánicas. Esto implica que haya que implementar ciertos elementos por duplicado, la primera creación aislada para comprobar el trabajo y después su implementación en el proyecto incremental. Aunque parezca innecesario hacerlo de esta manera, ya que gran parte de elementos hay que hacerlos por duplicado, es beneficioso a la hora de estar aprendiendo a desarrollar una plataforma nueva, ya que la segunda vez que se implementa, se optimizan los elementos tanto a nivel de funcionamiento como de código.

El desarrollo del trabajo fue secuencial en cuanto a los niveles. Para empezar, se ha desarrollado el nivel de *Hack 'n' Slash*, debido a que era el más completo a la hora de desarrollo de mecánicas, configuración y adaptación de mecánicas, modelado de elementos y creación de una inteligencia artificial para los enemigos.

Una vez completado el género de *Hack 'n' Slash*, se ha seguido desarrollando el género de combate por turnos. Este género proponía un reto debido a que el jugador no es el encargado de controlar un solo enemigo, así como la *IA* no se encargaba de manera un solo enemigo. En este caso, los controladores de personajes, tanto jugador como *IA*, tenían bajo su control diferentes personajes.

Por último, se ha desarrollado el género de sigilo. En este caso se ha desarrollado un sistema muy básico de sigilo donde el jugador es invisible para los enemigos si está oculto. Parte de las mecánicas básicas del sistema de sigilo se han cogido de otros géneros ya implementados, como pueda ser el movimiento por el mundo, o el control de la cámara.

Para ver un desarrollo en profundidad de los elementos implementados, están detallados en el apartado ***5.3. Desarrollo e implementación***, en este documento.

4.1. Control de tareas

Para el control de tareas pendientes y en proceso, se utilizará la herramienta *Trello*. Esta herramienta se basa en una simulación de un tablero, donde a modo de notas se ponen las tareas en diferentes categorías, como se muestra en la ***ilustración 8***. *Trello* tiene dos modalidades de uso, la versión gratuita que no limita la creación de tableros e inclusión de gente en los tableros, y la versión *Gold*, que añade ciertos elementos a los tableros, como aumento de espacio para las tarjetas. Para este proyecto, con la versión gratuita es suficiente.

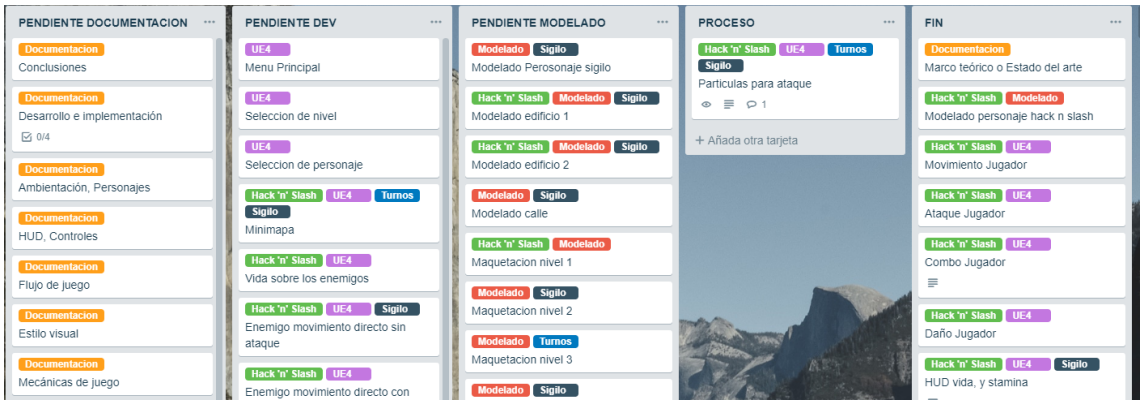

Ilustración 8. Tablero en Trello. Fuente: Elaboración propia.

4.2. Control de tiempos

Para el control de tiempos de cada tarea se ha utilizado la herramienta *Toggl*. Esta herramienta es básicamente un cronómetro, donde se marcará la tarea que se está realizando, o que se ha realizado para contabilizar el tiempo, como se muestra en la ***ilustración 9***. Los resultados de la medición de los tiempos están detallados en el apartado ***6.1 Control de tiempos en Toggl*** en la sección de conclusiones en este documento.

Los puntos fuertes de esta herramienta es que se pueden realizar equipos de trabajo y espacios de trabajo. Con todos estos parámetros se generan semanalmente informes del tiempo realizado durante ese tiempo.

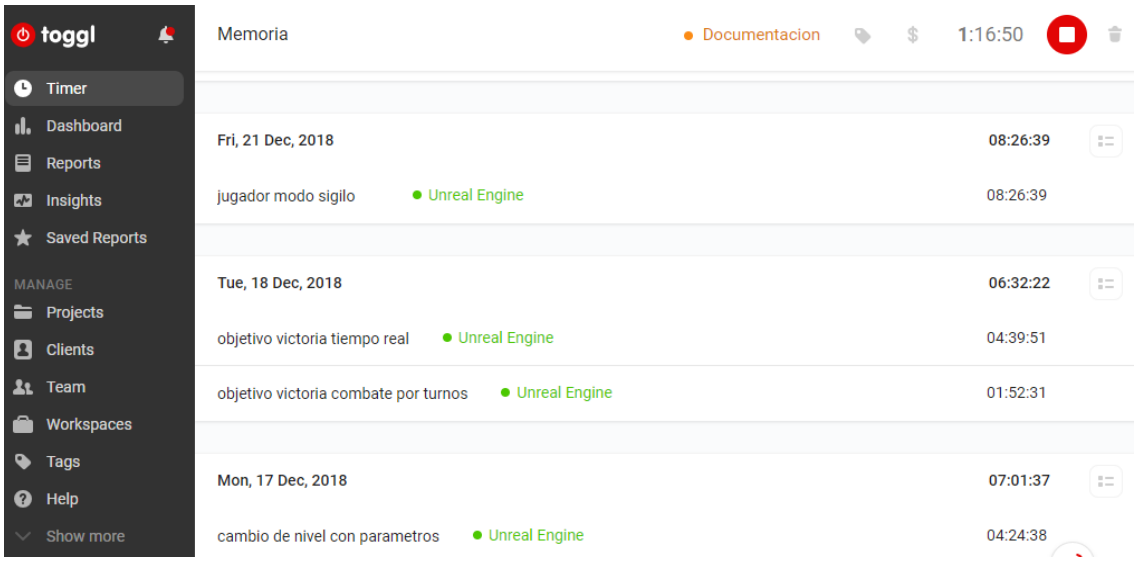

Ilustración 9. Tablero de Toggl. Fuente: Elaboración propia.

Hay varias modalidades de uso de esta aplicación, *Free*, *Starter* y *Premium*. Para un uso personal no es necesario más que el registro gratuito en la aplicación, ya que la mayor diferencia entre las tres modalidades es la gestión de equipos de trabajo y generación de informes.

4.3. Control de versiones

Para no perder el progreso y seguimiento de los cambios en el código del proyecto, todo este, esta subido a un repositorio donde se guardarán cambios, y se podrá volver a una versión anterior en caso de que haya algún error, o haya que recuperar algún cambio eliminado con anterioridad. El repositorio que se ha creado está alojado en *GitLab*. Esta plataforma permite crear ilimitados repositorios donde alojar nuestros proyectos, y a diferencia de otras plataformas con la misma finalidad, permite crear repositorios privados donde el creador o conjunto de creadores puedan acceder al repositorio.

Para la gestión de subida de versiones hay varias formas de hacerlo. En este caso se ha elegido la herramienta *GitKraken* ya que posee todas las funcionalidades necesarias para este caso y posee una interfaz simple y manejable como se puede observar en la ***ilustración 10***.

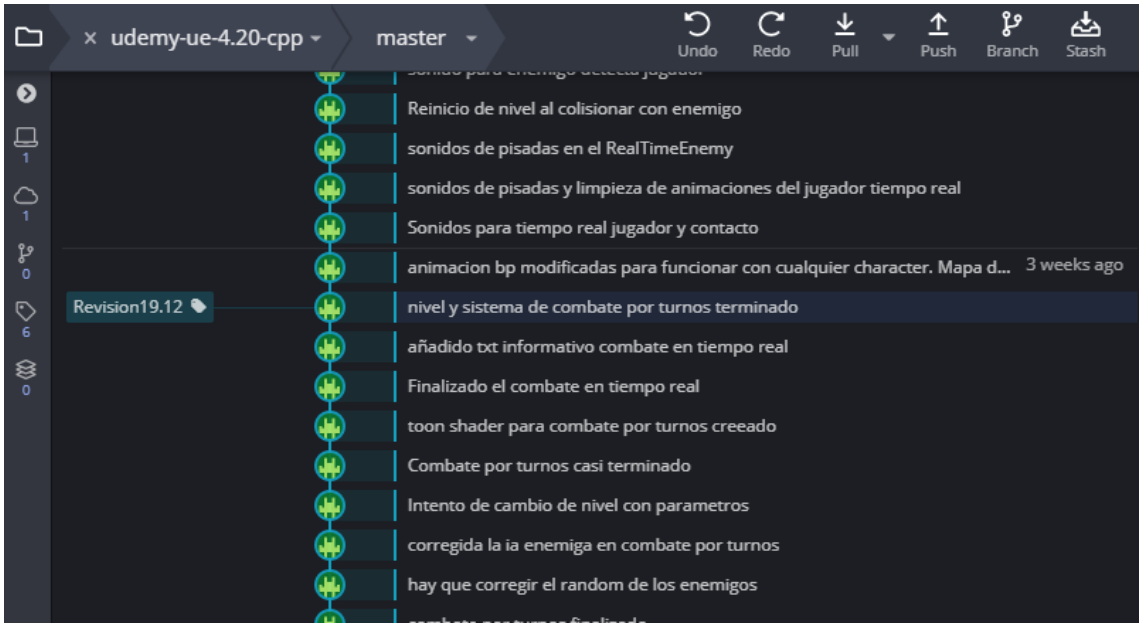

Ilustración 10. Seguimiento del repositorio con GitKraken. Fuente: Elaboración propia.

5. Cuerpo del trabajo

Este apartado del documento contiene detalles sobre los elementos relacionados con herramientas de desarrollo del proyecto, un análisis sobre los motores gráficos que actualmente son más utilizados tanto en grandes empresas, como en estudios pequeños. Además, se desarrollará un documento de diseño del videojuego, donde se plasmarán el concepto del videojuego a crear, que en este caso es un videojuego multigénero.

Se van a detallar los géneros a implementar, y de qué manera va a ser el flujo del videojuego una vez el usuario entre a jugar. Se hará un análisis de público objetivo con las pautas que posee la plataforma *PEGI (Pan European Game Information)*, así como un análisis del estilo visual que se ha implementado en el videojuego. Posteriormente, y antes de entrar en temas de cómo ha sido implementado, se definirán las mecánicas que posee el videojuego, así como los elementos en la interfaz gráfica que verá el jugador durante la partida.

En último lugar se encuentra el apartado de Desarrollo e implementación se detallan todos los elementos significativos del proyecto creado. Este desarrollo está categorizado en los diferentes géneros implementados en el videojuego.

5.1. Análisis de herramientas de desarrollo

En este apartado se va a realizar un análisis de las herramientas que se podrían utilizar para el desarrollo del proyecto. Se explicará qué es un motor gráfico, así como se analizarán distintos motores gráficos que hay actualmente en el mercado. También se detallará el coste que podría tendría la tarea de desarrollar un motor gráfico desde 0.

5.1.1. Motores gráficos

¿Qué es un motor gráfico? Un motor gráfico consiste en una aplicación (o conjunto de aplicaciones) que se encarga de la parte de comunicación entre el juego que se está desarrollando y los sistemas que se encargan de dibujar y gestionar recursos más cercanos a bajo nivel. Esto hace, que no haya que preocuparse en cómo va a dibujar un triángulo el ordenador, por ejemplo. Simplemente se le mandan al motor las instrucciones para que lo dibuje, y el motor se encargaría internamente en cómo hacerlo. Los motores actualmente

tienes muchísimas herramientas implementadas para ayudar al desarrollo de los videojuegos, como por ejemplo las herramientas gráficas para ver en tiempo real que es lo que se está haciendo, un sistema de físicas para simularlas, sistemas de audio para implementar el sonido y efectos en el juego, un sistema para hacer el juego multijugador online, etc.

5.1.2. Unity3D

Creado por *Unity* Technologies en 2005. Es uno de los motores que están en alza actualmente. El motor utiliza el lenguaje de programación *C#* y también es compatible con *JavaScript*, y se suele combinar con el entorno de programación de *Visual Studio*. Este motor no requiere demasiada potencia para iniciarse a la hora de desarrollar un proyecto y es por eso, por lo que pequeños estudios con bajo presupuesto suelen optar por esta opción. *Unity* tiene una buena comunidad de referencia para resolución de problemas y dudas. Además, a la hora de exportación a diferentes plataformas, *Unity* tiene soporte sobre casi todas las consolas que hay en el mercado, de manera que un único proyecto de videojuego, junto con una serie de retoques dependiendo de la plataforma y se obtiene un producto en las diferentes consolas del mercado, así como en PC y dispositivos móviles.

5.1.3. Unreal Engine

Creado por *Epic Games* en 1998. Este es un motor que saca gran rendimiento y potencia a los proyectos que se realizan en él. Se suele utilizar en grandes proyectos en los que se quiera obtener una gran calidad. Destaca por la implementación de su iluminación. Cuenta con dos maneras de desarrollo. Por un lado, se puede programar en lenguaje *C++* y por otro lado tiene un sistema de programación visual llamado *Blueprints* donde abstrayéndose de lo que podría ser la complicación de la programación se puede obtener

grandes resultados en los proyectos, aunque esto hace que pueda bajar el rendimiento respecto de la programación clásica. Actualmente las empresas grandes que desean realizar un juego y no quieren desarrollar un motor para dicho juego, suelen realizarlo en esta plataforma, ya que se obtienen grandes resultados con esta.

5.1.4. Motores implementados en C++

Hay diversos motores para realizar un juego que no tengan un entorno visual como lo puede tener *Unity* o *Unreal Engine*, o tengan herramientas externas para la visualización de diversos contenidos, como pueden ser *Ogre3D*, *Irrlicht*, *Godot*, etc.

Estos motores tienen un gran potencial, pero son menos utilizados en la actualidad, ya que se pueden obtener los mismos resultados en otras plataformas, y teniendo unas herramientas de visualización en tiempo real. El uso de estos motores podría suponer el aumento en el tiempo requerido para el desarrollo del videojuego ya que implica el mismo aprendizaje que pueda poseer *Unity3D* o *Unreal Engine* en cuanto a datos y funcionalidades específicos del lenguaje de programación que utiliza, así como el ajuste de elementos sin interfaz gráfica durante el desarrollo del proyecto.

5.1.5. Desarrollar un motor propio

Desarrollar un motor en *OpenGL*, *DirectX* o *Vulkan* es una opción que se descartó debido al extenso trabajo que requería este proceso. También uno de los elementos por los que se descartó realizar el motor, es que el objetivo del proyecto es realizar diferentes mecánicas, y realizar el motor desde cero habría conllevado minimizar este objetivo y habría quedado en segundo plano. Una de las mayores ventajas de realizar un motor, y es por las que la mayoría de las empresas grandes optan por este trabajo, es debido a que si haces el motor en su totalidad puede focalizar los puntos de optimización para que tu juego obtenga un rendimiento máximo. Por ejemplo, si se está desarrollando un juego donde se pretende tener un gran campo de visión, se pueden realizar diferentes técnicas para optimizar el uso de recursos con ese objetivo sin perder rendimiento, cosa que utilizando un motor ya desarrollado sería muy complicado.

5.1.6. Elección de motor gráfico

Una vez analizados todas las posibles herramientas que se podrían utilizar para el desarrollo del proyecto, se ha elegido el motor gráfico *Unreal Engine 4*, con su última versión 4.21. Esta elección se basa en varios puntos. En primer lugar, es una buena oportunidad para desarrollar sobre una plataforma que está en auge en estos últimos años en el sector de los videojuegos, debido al buen acabado que poseen los proyectos desarrollados. En segundo lugar, el desarrollo, así como la documentación necesaria para la realización, es un punto a favor para el análisis de tiempo que pueda poseer el proyecto para su aprendizaje desde el desconocimiento en el entorno y su acabado.

Por último, hay que destacar, el punto personal sobre el desarrollo del proyecto. Personalmente esta plataforma de desarrollo, la versatilidad que proporciona la combinación de programación clásica compilada en lenguaje *C++* junto con la programación visual por nodos hace que llame la atención a la hora de su elección.

5.2. Documento de diseño del videojuego (GDD)

5.2.1. Concepto

El videojuego desarrollado es un videojuego multigénero, esto es un videojuego que combina diferentes géneros dependiendo en un sistema de niveles aislados entre ellos en los que cada uno de los niveles posee mecánicas propias de los diferentes géneros que se han implementado. El objetivo de este este desarrollo es la simulación de un proyecto de gran amplitud, así como la combinación y reutilización de diferentes mecánicas a la hora de su desarrollo. La creación de este sistema hace que el jugador no entre en la monotonía de un sistema de mecánicas común durante toda la partida.

5.2.2. Géneros implementados

El juego tendrá una pantalla principal, donde el jugador selecciona el nivel que le toque dependiendo del progreso que tenga realizado. Una vez seleccionado el nivel, el género de juego del nivel, podrá ser uno de los siguientes:

Estilo de juego RPG por turnos: Con una cámara en tercera persona, el jugador se moverá libremente por el mapa y podrá mover la cámara con el personaje jugador como pivote de rotación. Situados en el mapa, tenemos una serie de enemigos colocados. En el caso de colisionar con estos enemigos, cambiará la escena actual a una escena de combate.

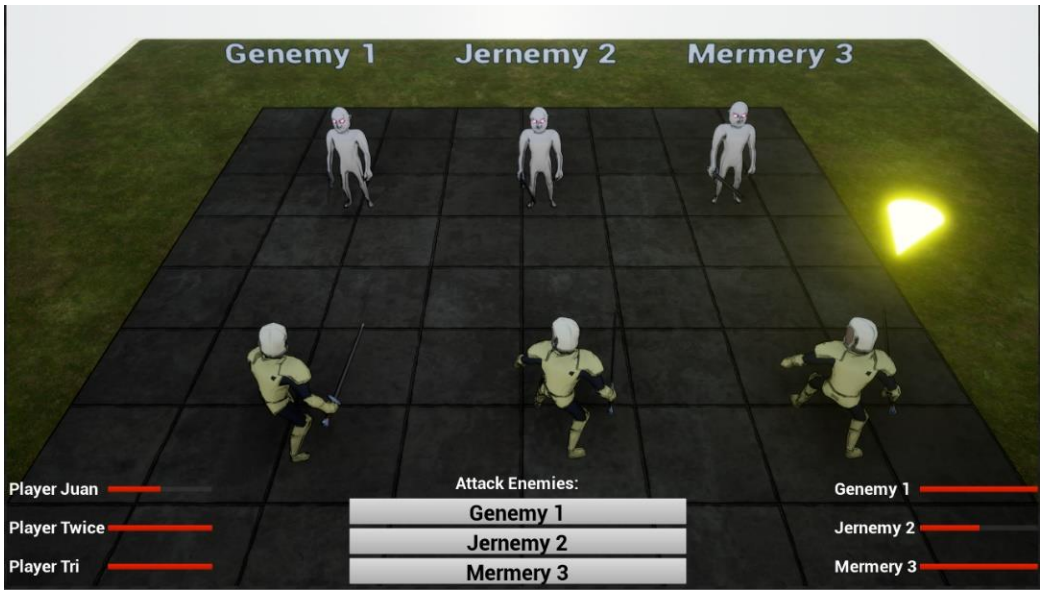

Ilustración 11. Escena de combate por turnos. Fuente: Elaboración propia.

En esta escena de combate tenemos dos bandos diferenciados, como se puede observar en la ***ilustración 11***. En la parte superior nos encontramos con los enemigos, y en la parte inferior se encuentran los jugadores que controlamos. Para diferenciar el jugador que posee el turno actual tendremos un indicador amarillo sobre el personaje.

Hack 'n' Slash: El movimiento a través del entorno en este género es similar al implementado en el combate por turnos. A diferencia de ese género, en este el jugador podrá atacar y encadenar ataques en tiempo real como se puede apreciar en la ***ilustración 12***. Los enemigos tienen una serie de sentidos de vista con un radio a su alrededor. En caso de entrar en ese radio enemigo, estos dejaran su movimiento actual para atacar al jugador. El objetivo de este mapa, al igual que el del género explicado anteriormente, será el de eliminar todos los enemigos del mapa.

Ilustración 12. Captura de combate Hack 'n' Slash. Fuente: Elaboración propia.

Sigilo: Para el género de sigilo se ha optado por una simplicidad en sus elementos como se puede observar en la ***ilustración 13***. Para añadirle un toque de dificultad, el jugador no podrá eliminar enemigos durante el desarrollo del nivel. Para no limitar la experiencia de juego, los enemigos no podrán detectar al jugador a menos que entre dentro de una luz que lo exponga. El objetivo de este mapa es encontrar el elemento para la conclusión del nivel. En caso de que un enemigo detecte al jugador ira a directo a por él. Si colisiona el enemigo con el jugador, este tendrá que volver a empezar el nivel.

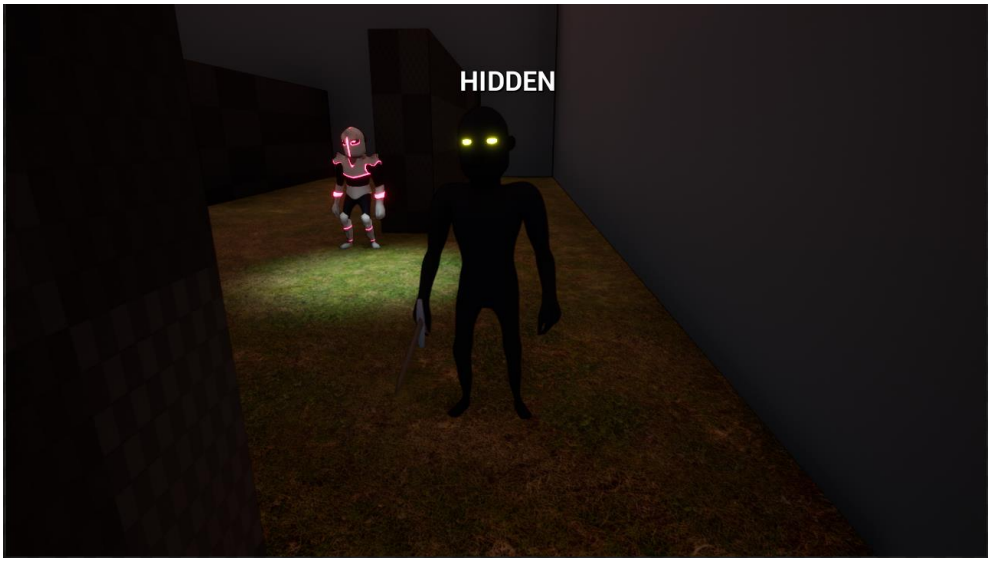

Ilustración 13. Captura de nivel de sigilo. Fuente: Elaboración propia

5.2.3. Propósito y público objetivo

El principal objetivo del juego es presentar un producto dinámico, con diferentes géneros de manera que cada nivel sea de una manera diferente, haciendo así que el juego no se convierta en una repetición de mecánicas, con diferentes escenarios, donde el jugador pueda aborrecer estas mecánicas.

El videojuego está dirigido a jugadores de un amplio rango de edades. Por ello, se apuesta por un sistema de partidas cortas y niveles cerrados.

Para el análisis del público objetivo se ha consultado el sistema de etiquetas de clasificación de *PEGI.* Para la clasificación de software digital, la plataforma tiene una serie de etiquetas que se puede ver en la ***ilustración 14***. Por un lado, tenemos los descriptores de contenido, estas son las etiquetas que marcaran posteriormente una clasificación de edad recomendada para el videojuego.

Ilustración 14. Etiquetas de la clasificacion PEGI. Fuente: https://www.pegi.info/es

Según la información que nos provee *PEGI* el juego sería calificado para mayores de 7 años ya que entraría dentro de la descripción provista: "*El contenido del juego con escenas o sonidos que pueden atemorizar a los niños más pequeños debería incluirse en esta categoría. Las formas muy suaves de violencia (violencia implícita, no detallada o no realista) son aceptables para un juego con una clasificación PEGI 7.*" (PEGI, 2018)

5.2.4. Estilo visual

El estilo visual del juego tendrá un estilo sencillo, no demasiado detallista para encajar con su carácter amigable. El estilo visual que más encaja con este concepto es el de dibujo animado o pintado a mano. Serán texturas simples, con colores vivos, y bordes marcados como se puede observar en la ***ilustración 15***. Para obtener este efecto de iluminación se implementará un efecto *Cartoon Shading* o *Cel Shading*.

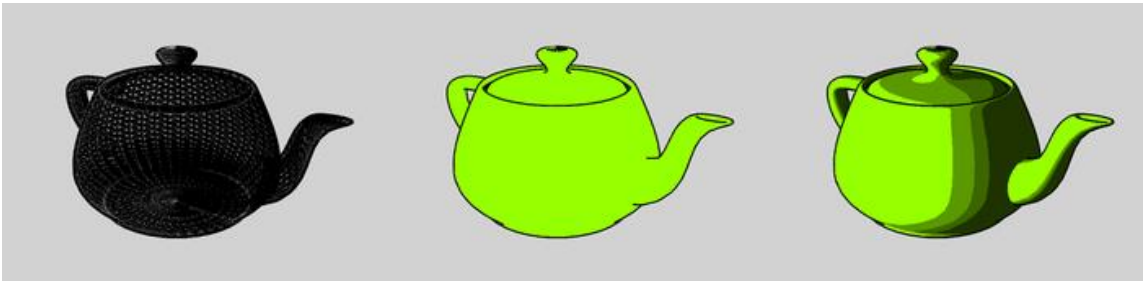

Ilustración 15. Composición para sombreado Cel Shading. Fuente: https://es.wikipedia.org/wiki/Cel_shading

La técnica de *Cartoon Shading* o *Cel Shading* consiste en la limitación de la difuminación del sombreado e iluminación en los elementos del juego, simulando así los dibujos animados. En vez de tener una atenuación realista de la iluminación haciendo un sombreado progresivo, se limita a diferentes pasos, de manera que la iluminación se asemeja a una serie de animación con unos cambios marcados en los diferentes niveles de sombreado como se puede observar en la ***ilustración 16***, donde podemos observar las diferencias entre una iluminación realista con un difuminado progresivo de las sombras, y un cambio marcado entre los elementos de sombreado.

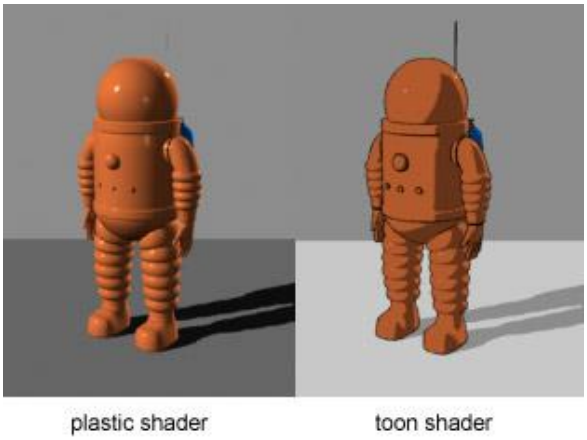

Ilustración 16. Diferencia entre realista y Cel Shading. Fuente: https://es.wikipedia.org/wiki/Cel_shading

5.2.5. Mecánicas de juego

En esta sección se va a desarrollar las diferentes mecánicas del juego en los diferentes géneros que posee. En primer lugar, se definen las mecánicas generales presentes en todos los géneros implementados. Posteriormente se analizarán las mecánicas propias de cada uno de los géneros.

5.2.5.1. Mecánicas generales

El videojuego posee diferentes mecánicas que son comunes para todos los géneros que tiene implementados. El mapa de juego será una escena en 3 dimensiones donde se podrá manejar el personaje a través del nivel y éste tendrá una cámara en tercera persona, donde la cámara seguirá al personaje y rotará alrededor de éste como punto de pivote.

En todos los mapas, los enemigos tendrán un sistema de detección para poder determinar si el jugador está cerca, pudiendo ser estos de comprobación de distancia, o un sistema sensorial de visión o ruido. En caso de que el jugador sea detectado el enemigo irá directo hacia la posición donde se ha detectado, en cuyo caso se perseguirá al jugador hasta que desaparezca del rango de visión o sea alcanzado. Dependiendo del género en el que se esté jugando habrá diferentes mecánicas en el caso de que estén en un rango cercano el enemigo y el jugador.

5.2.5.2. Mecánicas Hack ‘n’ Slash

Durante el desarrollo en este tipo de mapas tendremos una barra superior donde se indicará el nivel de vida y estamina del personaje. En caso de que la vida del jugador llegue a 0 se mostrará la pantalla de fin de partida.

El desarrollo del combate con los enemigos durante estos mapas será en tiempo real. Los enemigos detectarán cuando el jugador está a una distancia cercana y se dirigirán a atacarlo. Habrá asignado un botón para atacar, de manera que si se presiona repetidas veces realizará el personaje varios ataques a modo de combo. Con cada ataque disminuye la barra de estamina, y en caso de agotarse no podrá atacar más. Esta barra se rellena automáticamente con el paso del tiempo.

Resumen de mecánicas:

- Movimiento libre por el mundo.
- Al presionar el botón de ataque el jugador desencadena un ataque.
- A la sucesión de ataques el jugador realiza un combo de ataques.
- Los enemigos detectan por distancia donde está el jugador y van a por él.

5.2.5.3. Mecánicas Combate por turnos

En estos mapas se modifican los comportamientos de los enemigos. Una vez el jugador y el enemigo colisionen, se entrará en el mapa de batalla, donde nos encontraremos a un lado el equipo de jugadores que controlamos y a otro lado el equipo de enemigos que hay que derrotar. El sistema de combate se basará en una serie de turnos ordenados por un índice de acción, de manera que cada personaje en combate tiene su momento de acción o turno. El objetivo del combate será derrotar a todos los enemigos. En caso de que llegue a 0 la vida de todos nuestros personajes, nos habrán derrotado y se mostrará la pantalla de fin de partida.

Resumen de mecánicas:

- Movimiento libre a través del mapa del mundo.
- Al colisionar con un enemigo se cambia al mapa de combate.
- Durante el combate hay un sistema de turnos para cada participante.
- Durante el turno se podrá atacar a un enemigo.
- Al completar el combate se vuelve al mapa del mundo.

5.2.5.4. Mecánicas en Sigilo

El objetivo de estos niveles es ligeramente diferente al del resto de los niveles. Durante el desarrollo de este tipo de niveles, tendremos que superar el nivel sin que los enemigos nos detecten. Habrá una serie de puntos de control durante el nivel, y en caso de que los enemigos nos detecten y nos capturen (colisionen con el jugador) se mandará al jugador al punto de control. El jugador tiene un indicador en la pantalla que muestra el estado del

jugador, si está oculto o expuesto. En caso de estar oculto, los enemigos no podrán detectar al jugador.

Resumen de mecánicas:

- Movimiento libre a través del nivel.
- Evitar detección durante el desarrollo del nivel.
- Sistema de puntos de control.
- Reinicio al punto de control en caso de ser detectado.
- Evitar el sistema de sentidos de los enemigos estando oculto.

5.2.6. HUD

El *HUD (Head-Up Display)* son los elementos que se muestran en la pantalla referentes a los datos del videojuego, como pueden ser barras de vida, mensajes, objetivos, y otros tipos de indicadores dentro del juego.

Debido a que el juego se ha desarrollado con diferentes géneros, cada uno de estos tendrá una interfaz diferente durante el juego. Todos tendrán en común un mensaje en la parte inferior de la pantalla con los controles que tiene el jugador.

Por un lado, para el estilo *Hack 'n' Slash* se sitúa una barra de vida y estamina clásicos en este género, en la parte superior de la pantalla, como se muestra en la ***ilustración 17***. En la parte lateral derecha de la pantalla tenemos el resumen de objetivos del mapa, indicando los objetivos necesarios para completar el nivel.

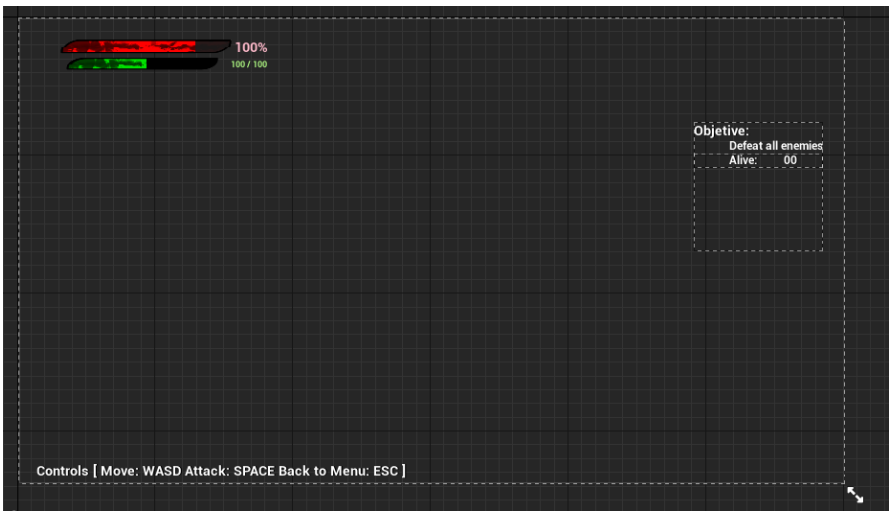

Ilustración 17. HUD para el género Hack 'n' Slash. Fuente: Elaboración propia.

Para el estilo de combate por turnos tenemos durante el combate los comandos que podremos realizar en nuestro turno, así como la vida de cada uno de los personajes que están en combate y los nombres identificativos de cada uno de los participantes del combate. Además, hay un indicador encima del personaje que tiene el turno actual, además de mostrar en cada cambio de turno una animación indicando quien posee el turno.

En cuanto al mapa de sigilo, la única interfaz que posee el nivel será un indicador en la parte superior donde se nos marca el estado que se encuentra el jugador: Oculto o Expuesto.

5.3. Desarrollo e implementación

En el siguiente apartado vamos a detallar cómo se han implementado las diferentes mecánicas que posee el videojuego. Cabe destacar que debido a los dos sistemas de desarrollo que tiene *Unreal Engine*, que son el lenguaje de programación clásico de *C++* o sistema de programación visual *Blueprints.* La filosofía que recomienda *Epic Games* a la hora de los desarrollos se basa en que la parte más compleja y pesada de cálculo recaiga sobre *C++* debido a la optimización que posee y después la asignación visual de elementos (como puede ser la asignación de la malla, o la animación que reproducir al recibir daño) sea configurada desde *Blueprints*. También hay que mencionar que se puede hacer ambas partes íntegramente desde *C++* o desde *Blueprints*, al final recae sobre el desarrollador o desarrolladores esta elección.

En mi caso, para el sistema de *Hack 'n' Slash*, se ha seleccionado desarrollarlo sobre la filosofía de *Epic Games*, cediendo el grosor de computación a *C++* y los aspectos visuales a *Blueprints*. Y posteriormente el desarrollo del sistema de combate por turnos será desarrollado íntegramente sobre *Blueprints*.

El objetivo de este cambio es comprobar la comodidad, optimización y velocidad a la hora de implementar las diferentes mecánicas que poseen ambos géneros.

Para facilitar la explicación de los diferentes términos, así como la implementación de los objetivos, se ha seguido un orden para ayudar la comprensión. En primer lugar, se

definirán los términos necesarios para comprender el ámbito de *Unreal Engine 4*: qué son los *Blueprints*, cómo funcionan y qué elementos se usan en el desarrollo y la creación de componentes y animaciones. Posteriormente se explicará que es el *Cel Shader* y cómo se ha implementado para que presente el efecto de dibujo deseado, así como la gestión de métodos de entrada dentro del sistema. Por último, se comentarán los diferentes elementos implementados categorizados por género.

5.3.1. Blueprints

Este término ha sido mencionado en bastantes ocasiones durante este documento, y se ha a aclarar qué es exactamente este sistema. El sistema de programación *Blueprint* es un sistema incluido en *Unreal Engine* para evitar abstraerse de ciertos elementos a la hora del desarrollo de un proyecto. Tanto como si se quiere desarrollar un proyecto íntegramente en *Blueprints* como si quiere hacer la asociación de elementos gráficos, es un sistema potente y muy flexible de usar.

El sistema de *Blueprints* funciona a partir de unos nodos que conecta el desarrollador durante la implementación de los diferentes elementos del juego. Estos nodos comúnmente son las funciones que se utilizan en el lenguaje de *C++*. De esta manera los parámetros que recibirían y los datos de salida que retornarían estas funciones se convierten en enlaces de entrada y de salida de los nodos. Para ayudar a la comprensión de esta explicación se van a explicar a continuación varios ejemplos de *Blueprints*.

En primer lugar, se realiza una comparación entre *Blueprint* y función en *C++* en la ***ilustración 18***. En esta imagen se encuentra la misma función de nodo en *Blueprint* en la parte izquierda, y la implementación de la función en *C++* en la derecha. Como se puede observar a la izquierda, el nodo recibe 4 parámetros de entrada y tiene un parámetro de salida. En el caso de la función de *C++*, consultando la documentación sobre ella, nos dice que será modificado el valor del parámetro *OutSweepHitResult*. Esto hace que la implementación en los dos sistemas para esta función sea la misma.

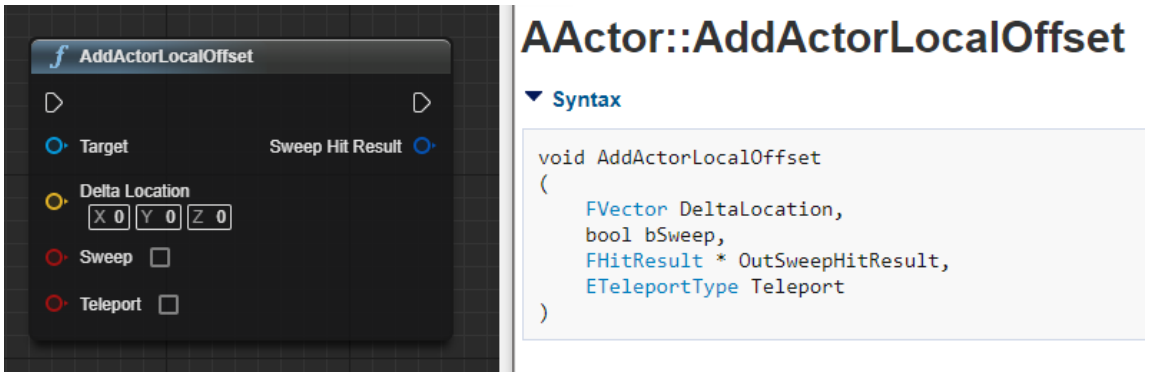

Ilustración 18. Implementaciones en Blueprints y C++ de la misma función. Fuente: https://api.unrealengine.com/

Después, para la relación y uso entre varios nodos se presenta la ***ilustración 19***. En esta se observan 2 nodos conectados, uno de ellos es un evento que se ejecuta cada ciclo de juego llamado *Tick* (en otros sistemas es llamado *Update*). Por otro lado, tenemos un nodo *AddActorLocalRotation* encargado de añadir rotación al actor asociado. Esto significa que con estos dos nodos se tiene un actor que rotará sobre sí mismo durante la ejecución del videojuego.

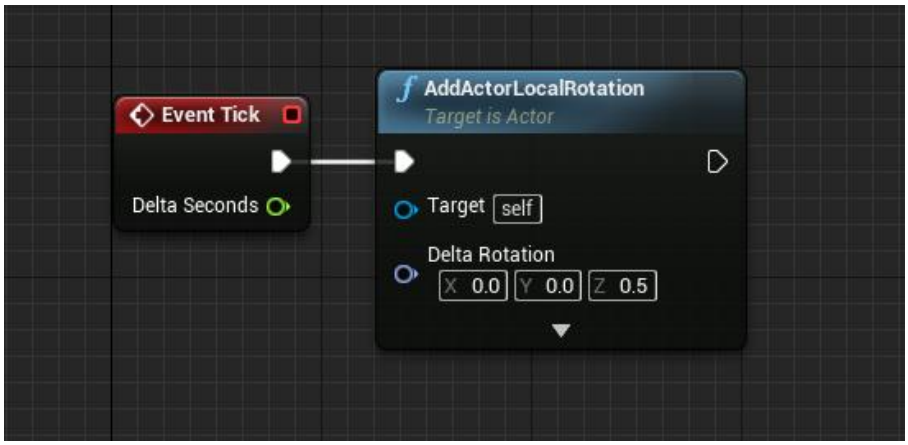

Ilustración 19. Blueprint de rotación de un elemento. Fuente: Elaboración propia.

Los *Blueprint* no solo se utilizan para la programación de elementos en el entorno. Hay distintos tipos de *Blueprints* dentro de *Unreal Engine*. Para la creación de los materiales que se renderizarán en el entorno, y que estarán asociados con los elementos 3D, se utilizan estos sistemas de nodos como se puede observar en la ***ilustración 20.***

Ilustración 20. Blueprint de material. Fuente: Elaboración propia.

En la ***ilustración 20*** se pueden observar diferentes nodos conectados entre sí en unas zonas blancas situadas en la parte superior a modo de título. Estas zonas blancas son zonas comentadas para ayudar al desarrollador a leer los *Blueprint*. Por otro lado, los materiales tienen un nodo de salida, que sería el nodo en la derecha representado con gran número de puntos de entrada. Todos los puntos de entrada resaltados en blanco podrían ser utilizados en este caso. Dependiendo de la finalidad del material, el nodo tendrá diferentes puntos de entrada activados para su utilización.

En último lugar, hay que destacar otro tipo de *Blueprint* diferente, el *Blueprint* de animación. Estos *Blueprint* son los encargados de asociar una animación a una malla de un personaje, dependiendo de ciertos parámetros. Normalmente estos parámetros son unos estados que van cambiando para animar un personaje de una manera u otra según se hayan programado.

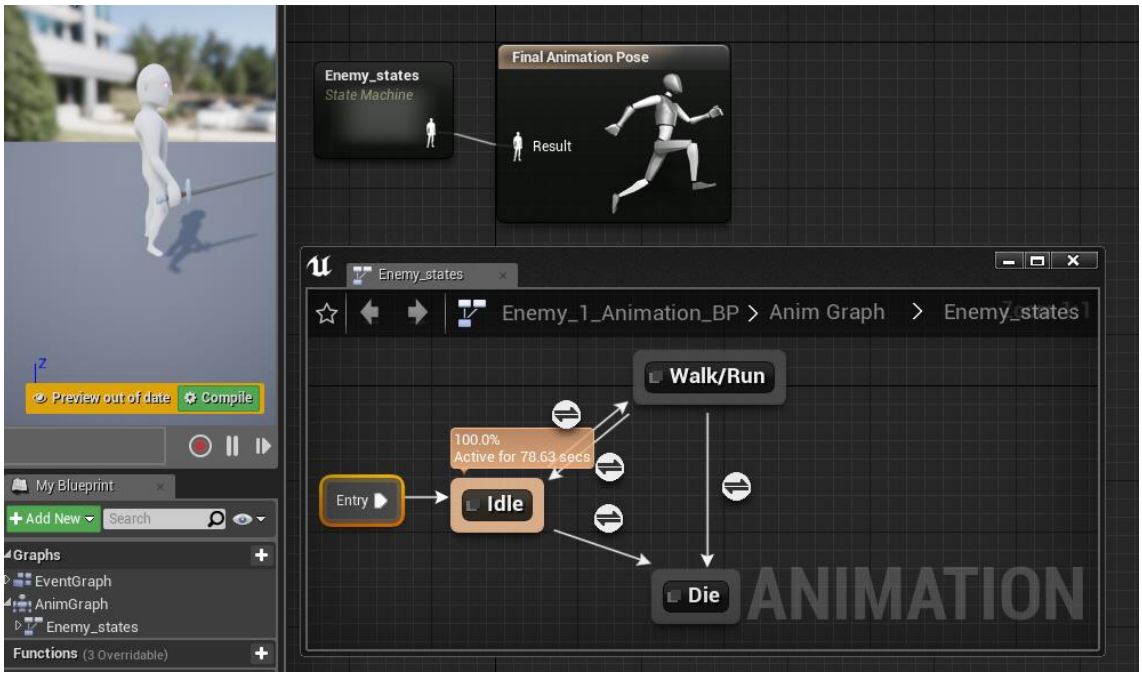

Ilustración 21. Blueprint de animación junto con sus estados. Fuente: Elaboración propia.

En la ***ilustración 21*** se presentan dos secciones. Por un lado, tenemos la sección superior donde está el nodo de la máquina de estados para reproducir una animación que está conectado con el nodo final de pose que es que encargado de activar una animación. Dentro del mencionado nodo de máquina de estados esta la sección inferior, donde se presentan diferentes estados con unas transiciones marcadas con un símbolo redondo blanco con dos flechas. Estas transiciones están programadas mediante nodos para activarse dependiendo de ciertas condiciones, como pueda ser el caso que el parámetro vida de nuestro personaje llegue a 0.

5.3.2. Funcionamiento de Unreal Engine

El funcionamiento básico en Unreal Engine se dividen en varios elementos que hay que definir antes de entrar en detalle sobre el desarrollo del proyecto. Los elementos definidos son: *Pawn*, *GameMode* y *Controller*. Debido al funcionamiento en componentes de *Unreal Engine 4*, todos los elementos que se manejarán y que podrán estar situados en la escena, son los llamados actores. Estos actores están diferenciados por los componentes que poseen. Durante el desarrollo del proyecto se han creado y utilizado diferentes

componentes. Este desarrollo está comentado en el apartado ***6.3 Creación de componentes***.

Para ayudar la lectura y compresión del funcionamiento está la ***ilustración 22***, donde en forma de diagrama tenemos los elementos básicos en el funcionamiento de *Unreal Engine*. En esta ilustración se representa el juego junto con un *GameMode* y *GameState* dentro de éste. Estos dos elementos se podrían denominar como el sistema de reglas y el sistema de persistencia de datos entre niveles. Conectado con el juego está el *PlayerController* que es el controlador encargado de traducir las órdenes del jugador dentro de la partida. Este controlador, como se muestra en la parte izquierda de la imagen, contiene una interfaz de juego que se observara en la pantalla o *HUD*, la interacción con elementos de entrada o *Input* y un gestor de cámara que se denomina *PlayerCameraManager*.

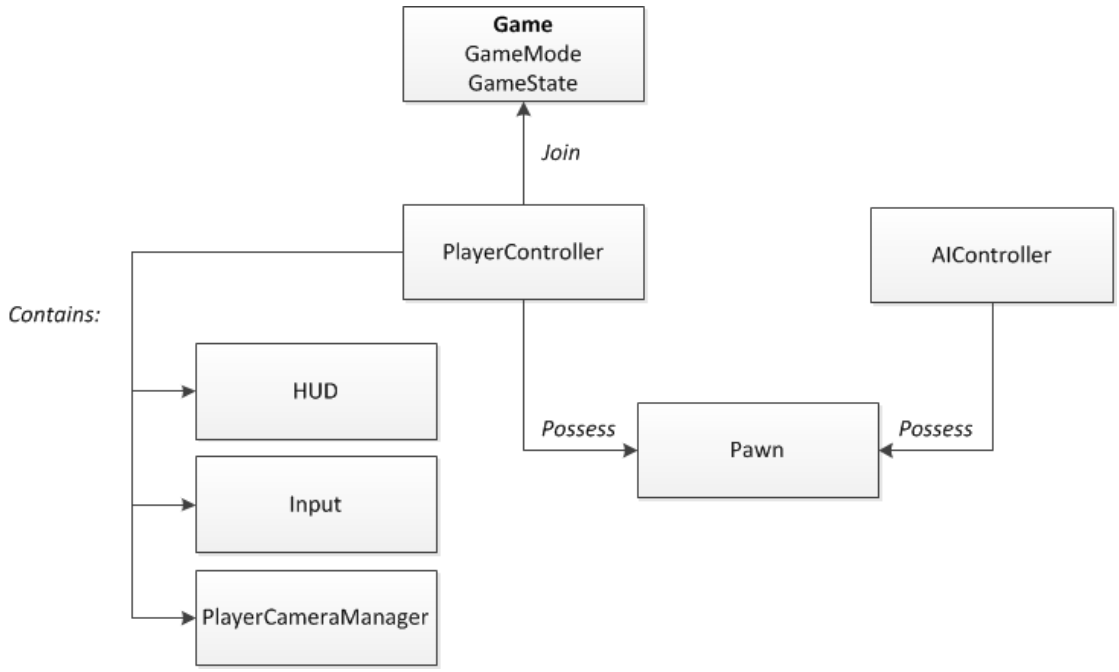

Ilustración 22. Diagrama del funcionamiento de Unreal Engine. Fuente: https://docs.unrealengine.com/en-us/Gameplay/Framework/QuickReference

Siguiendo con las indicaciones de la imagen, se puede observar que se presentan dos controladores diferentes. Por un lado, está el *PlayerController* y por otro el *AIController*, y ambos están marcados con la indicación de que poseen al elemento *Pawn*. Esto significa

que los *Pawn* son los actores pueden ser controlados tanto por el jugador o como por una *IA*.

En los siguientes puntos dentro de este apartado se definen con más exactitud los elementos comentados.

5.3.2.1. Pawn

Los objetos de tipo *Pawn* es la clase base de un actor para poder ser controlado. El controlador o *Controller* de este actor no tiene que ser obligatoriamente un jugador, puede controlarlo una inteligencia artificial (*IA*) dentro de nuestro juego. El *Pawn* es la representación de un elemento controlado en la escena, pero eso no significa que este elemento tenga que tener una representación visual. Por ejemplo, dentro del género de combate por turnos implementado en este proyecto, el *Pawn* que se utiliza es un gestor de turnos que se encarga de dirigir y manejar todos los personajes dentro del combate. Este gestor de turnos no tiene ningún modelo 3D asociado que se aprecie en la escena.

5.3.2.1. Controller

Los *Controller* son actores sin representación en la escena encargados de manejar un actor de la clase *Pawn* o derivados de esta. En el caso que el controlador sea una *IA* para el funcionamiento de los enemigos, se estaría hablando de la clase *AIController*.

El funcionamiento de los controladores, para el manejo de los *Pawn* que controlan es muy sencillo. Una vez se posee un *Pawn* el controlador recibe las notificaciones de eventos que reciben el actor que se está poseyendo, de esta manera, se programan las respuestas a los eventos según el desarrollador lo necesite.

Suponiendo el caso que un *Pawn* esté siendo controlado por un *AIController* dentro de nuestro juego. Para implementar la eliminación de un enemigo, se programa el *AIController* para comprobar en el evento de recibir daño del sistema. Si la vida es igual o inferior a 0, este controlador será destruido de la escena y se le mandará un mensaje al actor que estaba siendo controlado que active el evento de ser destruido.

5.3.2.2. GameMode

Este elemento dentro del funcionamiento del juego se encarga de manejar la información importante sobre el transcurso de la partida. El *GameMode* es el encargado de definir reglas para el transcurso de la partida, como pueden ser, la cantidad de jugadores permitidos, la condición de victoria o de derrota de los jugadores, etc.

Este elemento es programado como cualquier otro elemento de nuestro proyecto, pero hay una diferencia. En cada nivel siempre tiene que haber activo un *GameMode*, y éste puede ser común para todos los niveles, incluso si se desea desarrollar un *GameMode* diferente para cada nivel. En el caso concreto de este proyecto, al ser un sistema multigénero con diferentes condiciones de victoria y derrota dentro de cada nivel, se han desarrollado 3 distintos *GameMode* dentro del proyecto.

5.3.3. Creación de componentes

Debido a que el desarrollo y estructuración del motor está hecha por componentes, se han desarrollado dos componentes propios para la gestión de recursos del jugador durante la partida. Como se puede observar en la ***ilustración 23*** estos componentes han sido añadidos al listado de componentes que conforman el personaje jugador del género *Hack 'n' Slash*.

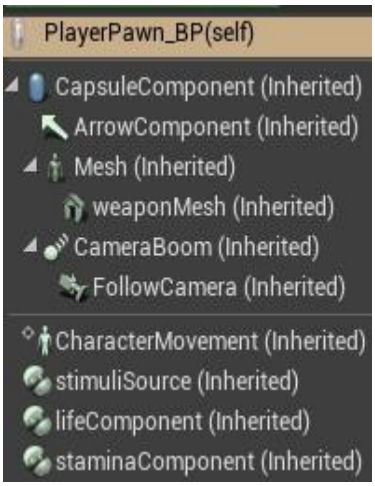

Ilustración 23. Listado componentes del actor PlayerPawn. Fuente: Elaboración propia.

Se han creado dos componentes *HealthComponent* y *MagicComponent*. Para la creación del componente es necesario que el componente herede de una clase base que nos proporciona *Unreal Engine 4* que sería el componente *UActorComponent.* Una vez creada, se definirán diferentes atributos para su gestión, un nivel máximo, un nivel actual, un porcentaje y el valor anterior que ha tenido el recurso. Además, se implementarán una serie de funciones y eventos para que puedan ser llamadas desde los *Blueprints* que se desarrollan y de esta manera obtener diferentes elementos.

```
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite) float magic; // Actual magic points
    UPROPERTY(EditAnywhere, BlueprintReadWrite) float fullMagic; // Max amount of magic
    UPROPERTY(EditAnywhere, BlueprintReadWrite) float magicPercentage; // magic / full magic
    UPROPERTY(VisibleAnywhere, BlueprintReadWrite) float previousMagic; // To smooth decreasing magic
    UPROPERTY(VisibleAnywhere, BlueprintReadWrite) float magicValue;
    UPROPERTY(VisibleAnywhere, BlueprintReadWrite) float magicPerAttack; // Amount of magic use when attack
    UPROPERTY(EditAnywhere, BlueprintReadWrite) UCurveFloat *magicCurve;
    UPROPERTY(VisibleAnywhere, BlueprintReadWrite) bool bCanUseMagic;
    UPROPERTY(VisibleAnywhere) FTimeline magicTimeline;
    FOnTimelineFloat timelineCallback; // Runs to reduce magic value
    FOnTimelineEventStatic timelineFinishedCallback; // Stops que reduction magic value
    float curveFloatValue;
    float timelineValue;

    UFUNCTION(BlueprintPure) float GetMagic();
    UFUNCTION(BlueprintPure) FText GetMagicIntText();
    UFUNCTION() void UpdateMagic();
    UFUNCTION() void SetMagicValue();
    UFUNCTION() void SetMagicState();
    UFUNCTION() void SetMagicChange(float value);
```

Ilustración 24. Declaración de atributos en MagicComponent y C++. Fuente: Elaboración propia.

Como se puede observar en la ***ilustración 24***, para que los eventos y funciones que se desarrollan sean accesibles posteriormente desde el editor de *Blueprints* de *Unreal Engine*, es necesario utilizar varios elementos. En primer lugar, para que los reconozca el editor es necesario que tengan unas directivas a forma de etiqueta o macro que son *UPROPERTY* y *UFUNCTION* dependiendo de si es una propiedad o una función. Dentro de estas macros se pueden definir unas etiquetas como se observa en la ***ilustración 23***.

Dependiendo de la etiqueta que se utilice se tendrá un acceso diferente posteriormente, serán clasificados los elementos dentro de ciertas categorías y demás opciones posibles. Una vez compilado correctamente, la única condición necesaria será agregar el componente a un actor que lo vaya a usar, ya sea desde código o desde el editor de *Unreal Engine*.

Para agregarlo desde el editor hay que seguir unos sencillos pasos. Primero es necesario tener una clase actor o una derivada en la que se le puedan agregar los componentes deseados, ya que no todas las clases pueden tener todos los componentes. Por ejemplo, las clases *Controller* no pueden tener el componente de fuente de estímulos (*StimuliSource*) ya que estos actores no pueden detectar estos elementos.

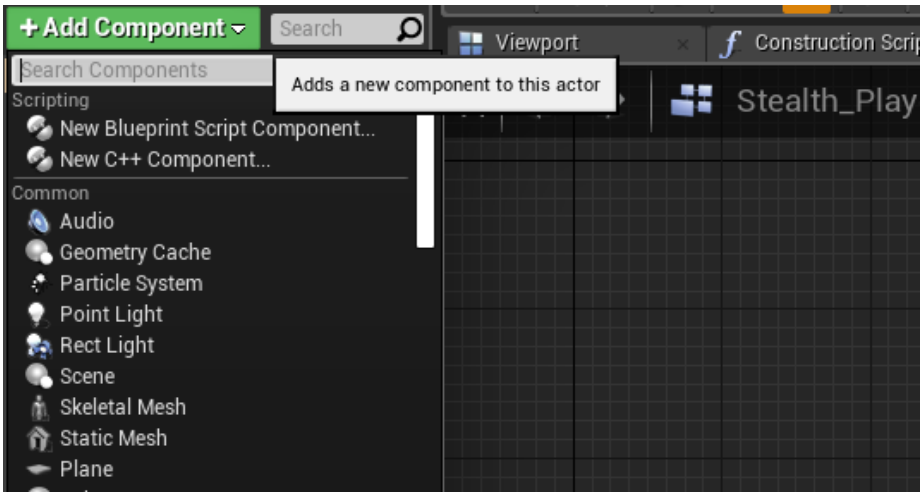

Ilustración 25. Listado de componentes para agregar. Fuente: Elaboración propia.

En la ***ilustración 25*** está definido el ejemplo de agregar un componente a un actor desde el editor de *Unreal Engine*. Dentro de la pantalla de edición del actor, está situado el botón "*Add Component"*. Una vez se pulsa, aparece el listado de componentes accesibles para ese actor.

5.3.3.1. HealthComponent

Este componente ha sido creado con la finalidad que todos los actores que puedan recibir daño lo reciban a través de este componente. En el género *Hack 'n' Slash* lo poseen actores como es el jugador y los enemigos. En el género de *RPG* con combate por turnos lo poseen todos los participantes de los combates.

En la ***tabla 1*** se reflejan los diferentes métodos implementados en estos componentes junto con su descripción.

Tabla 1. Métodos de HealthComponent.

Nombre del método	Descripción
BeginPlay	Evento que se ejecuta al iniciar la partida. En esta función se inicializan las distintas variables que posee el componente.
GetHealth	Este método devuelve el porcentaje de vida que posee el componente.
GetGealthIntText	Este método es el encargado de formatear los valores de la vida del componente para mostrarlo como un texto con el porcentaje restante de vida.
UpdateHealth	Este método es el encargado de actualizar los valores de la vida del componente. Este método es el único que puede modificar la vida del componente.

5.3.3.2. MagicComponent

Este componente ha sido creado con la finalidad que cualquier actor que requiera de algún tipo de gasto de un recurso que se agote y recargue de manera repetida utilice este componente. Para el jugador del *Hack 'n' Slash*, se utiliza este componente con el objetivo de simular un sistema de cansancio para limitar el uso de los combos de ataque durante el juego. La definición de los diferentes elementos que forman este componente están plasmados en la ***tabla 2***.

Tabla 2. Métodos de MagicComponent.

Nombre del método	Descripción
BeginPlay	Se inicializan las variables que posee el componente. Se instancia una curvatura para la transición de los valores.
TickComponent	Este método se encarga de actualizar los valores del componente en cada ciclo de juego. Esto simula una transición progresiva de los valores.
GetMagic	Método que devuelve el valor actual del recurso.

Nombre del método	Descripción
GetMagicIntText	Devuelve el texto formateado para mostrar el valor del recurso junto con el valor máximo posible.

5.3.4. Gestión de animaciones

La animación de elementos dentro de un videojuego es uno de los puntos visuales que más impactan durante el desarrollo de la partida. En el desarrollo de este proyecto se han creado modelos en 3D para su posterior animación. Debido a la cantidad de trabajo que requiere la animación de un personaje para su utilización en un videojuego, se ha optado por la utilización de las animaciones que nos proporciona la plataforma *Mixamo* de forma gratuita.

Esta plataforma permite utilizar un modelo humanoide que se haya creado por un programa de modelado 3D externo y subirlo a su plataforma online. Una vez subido el modelo, se presenta una biblioteca con una gran cantidad de animaciones para el uso del proyecto y su implementación en este caso en *Unreal Engine*. Para la gestión de las animaciones en *Unreal Engine* tenemos varios elementos que hay que configurar.

En la ***ilustración 26*** se presentan los elementos que se han utilizado. Por un lado, está el modelo en 3D junto con el esqueleto que enlaza los puntos claves de la malla con unos puntos de referencia. El objeto de físicas de la malla, donde se indica las colisiones que tendrá esa malla de una manera visual. Además, se presentan dos elementos más, el *Blueprint* de animación y el *Blendspace*.

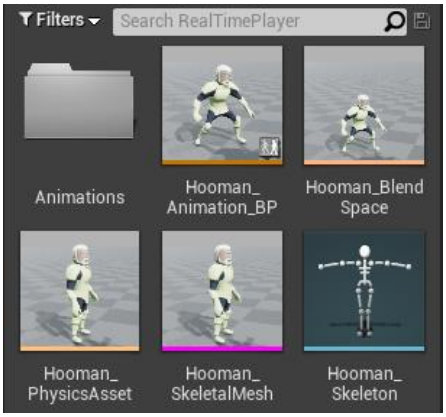

Ilustración 26. Elementos para las animaciones. Fuente: Elaboración propia

Comenzando con el *Blueprint* de la animación, este elemento es donde se crean y se les dan valor a ciertas variables del personaje, para poder utilizarlas en el gráfico de la animación. También se reciben notificaciones desde la animación para ejecutar diferentes funciones. En la ***ilustración 26***, se puede observar cómo se inicia todo el proceso con el nodo *Event Blueprint Update Animation.* Este es el evento estándar que se ejecuta cada ciclo en la animación para hacer comprobación de datos y de condiciones. En este caso, se actualiza la velocidad a la que se mueve el personaje, obteniendo la longitud del vector velocidad y también se comprueba si hay que actualizar el indicador para marcar el personaje como muerto, obteniendo la cantidad de vida restante en su componente *HealthComponent*.

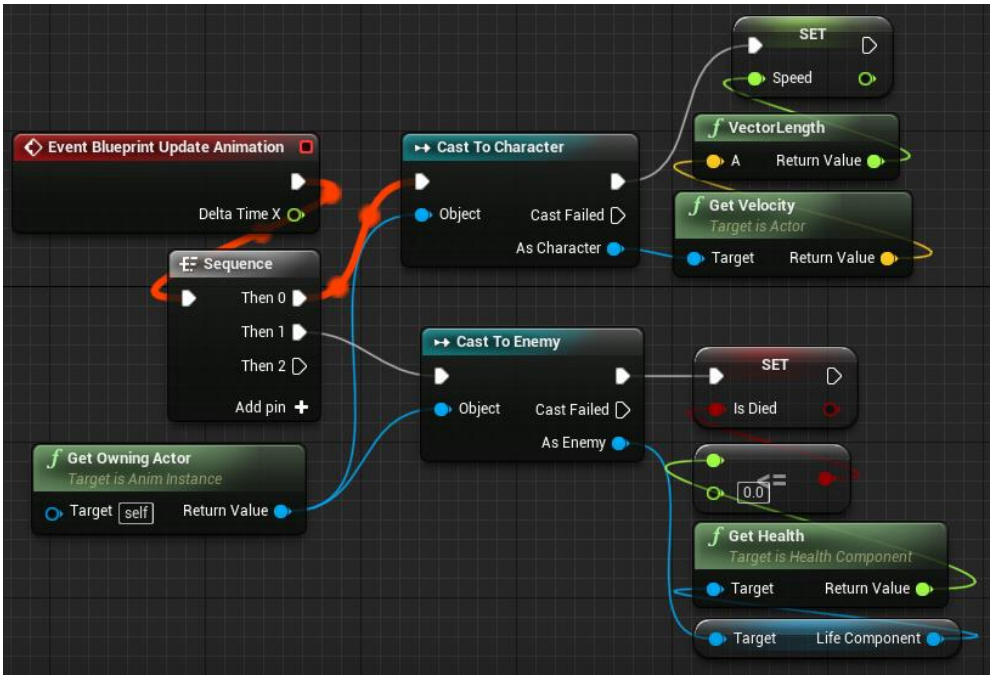

Ilustración 27. Blueprint de animación de un enemigo. Fuente: Elaboración propia.

Como se comentó con anterioridad, todos estos elementos y datos actualizados sirven para reproducir una animación dependiendo de una máquina de estados. Para ello, en otra parte del *Blueprint* de animación, dentro del grafo de animación, se configura la postura o pose que tendrá el modelo. En el caso de los actores configurados en este proyecto, se les ha creado un sistema de reproducción de huesos de manera que se puedan reproducir

distintas partes de las animaciones a la vez. En la ***ilustración 28*** se presenta la implementación de este sistema. El funcionamiento es sencillo. Se utilizan una serie de nodos cache para almacenar la pose que se va a reproducir y se analizan por un nodo de reproducción de animación por capas. En este nodo se configura un hueso del esqueleto de la animación que será el encargado de marcar la diferencia entre las dos capas que se estén reproduciendo. En este caso se ha configurado el hueso localizado en la cintura de manera que los personajes reproduzcan la animación de atacar y la de movimiento a la vez.

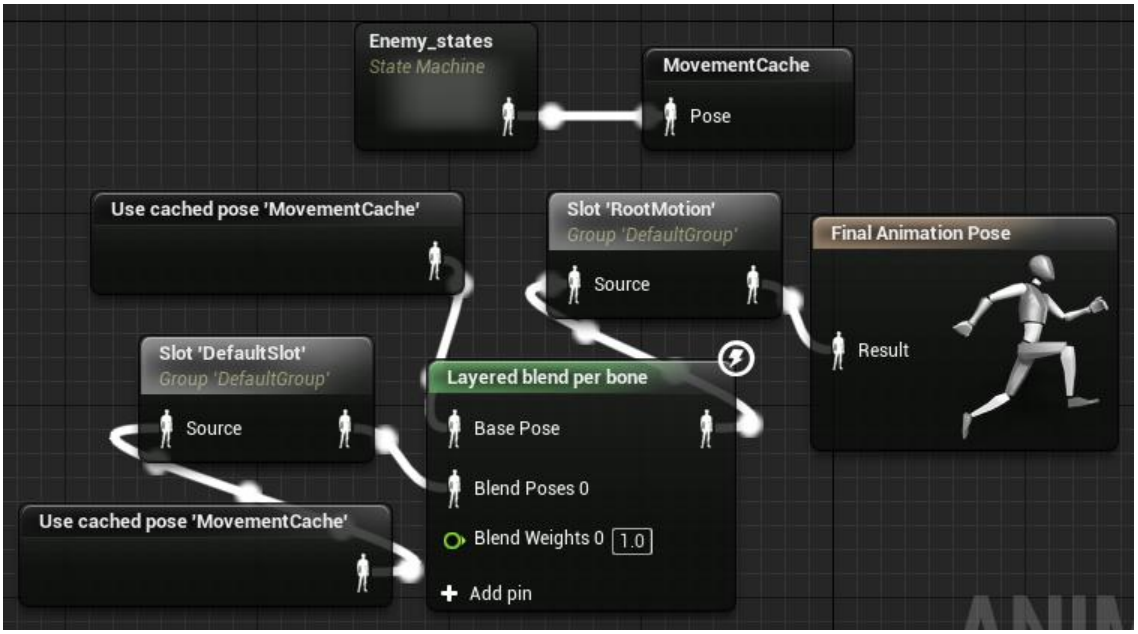

Ilustración 28. Gestión de estados de animación. Fuente: Elaboración propia.

Por otro lado, se configura el *Blendspace*. Este elemento presenta un gráfico donde se especifican los ejes a que representan, en este caso el eje X representa la dirección y el eje Y representa la velocidad.

Las animaciones se posicionan en un punto del grafico donde se reproducirán de forma íntegra. En los valores intermedios *Unreal Engine* se encarga de realizar una interpolación de las animaciones. Esto hace que la transición entre diferentes animaciones no tenga un cambio brusco y de la sensación de mayor naturalidad en los movimientos.

En la ***ilustración 29*** se muestran los elementos propios de las animaciones que se han desarrollado. Están los clips de animaciones o *Animation Sequence* representan la

animación con el añadido que se pueden configurar una serie de notificaciones o elementos para detallar la animación. A partir de estos clips se pueden crear los montajes de animaciones o *Animation Montage.*

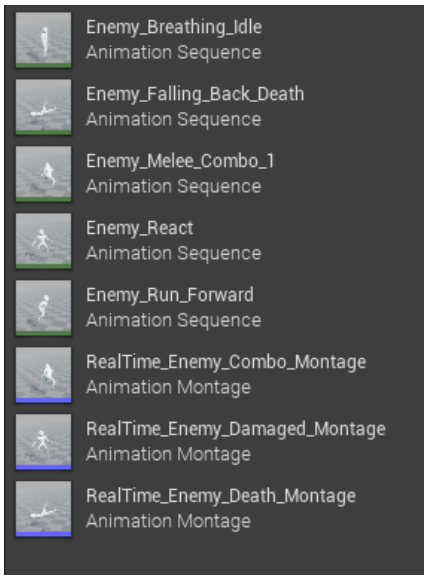

Ilustración 29. Clips de animación para un enemigo de Hack 'n' Slash. Fuente: Elaboración propia

Estos elementos representan una serie de animaciones que pueden ser seccionadas en distintos segmentos de manera que dependiendo de la situación se reproduzca un segmento. Por ejemplo, teniendo una secuencia de animación de salto completo, se secciona el clip en 3 partes. El inicio del salto, la suspensión en el aire, y la caída al suelo. Teniendo estas secciones se reproducirá el inicio del salto una vez el personaje reciba la notificación de iniciar el salto (por que el jugador haya pulsado la tecla). Una vez terminada esa sección, se reproducirá en bucle la sección de suspensión en el aire hasta que el personaje reciba la notificación que haya tocado el suelo, en cuyo caso se reproducirá la última sección del montaje.

5.3.5. Cel Shader

La obtención de este efecto de dibujo deseado se puede realizar de varias maneras. Por un lado, se puede hacer creando todos los materiales de cada elemento de la escena con

este estilo. De esta manera si se quisiera combinar un efecto *cartoon* con elementos realistas se podría hacer con este método. El problema que reside en este método es cuando no se desea combinar ambos elementos. Para conseguir este efecto, utilizando materiales aislados con efectos *cartoon* hace que haya que crear todos los materiales específicamente de esta manera. Hay otro sistema para crear el efecto *cartoon* dentro del juego. Este método se basa en crear un material de postprocesado de la escena llamado *shader* o sombreador.

Los *shaders* tienen como función base calcular los niveles de luz, oscuridad y color dentro de una imagen. Estos elementos trabajan sobre cada uno de los pixeles de la pantalla, lo que hace que sea un elemento altamente importante dentro del desarrollo del juego, ya que un fallo de optimización de este elemento hace que baje el rendimiento del programa de manera desastrosa.

El funcionamiento de *Cel Shader* en este caso es muy sencillo. Ya que el efecto deseado es un cambio brusco de sombras y resaltado de líneas, basta con buscar donde se encuentra el borde de los elementos. Para ello lo que se hace es comprobar los pixeles colindantes a los que se están analizando y se hace una media de los valores de color que tienen estos. En caso de encontrarse en la silueta de algún elemento, esta media hará que tenga un valor busco y lo resaltará.

Como se puede observar en la ***ilustración 30*** este proceso no solo resalta las líneas exteriores de los elementos, también resalta las líneas interiores. En el caso de este videojuego es un detalle aceptado dentro del efecto deseado.

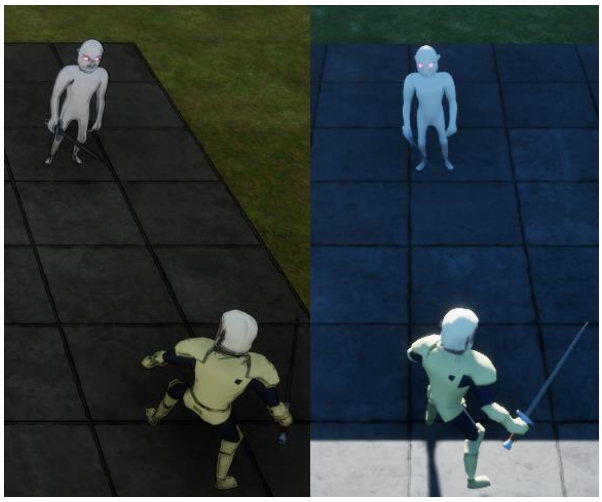

Ilustración 30. Diferencia entre shader cartoon y realista. Fuente: Elaboración propia.

5.3.6. Gestión de métodos de entrada

El sistema de gestión de métodos de entrada dentro de *Unreal Engine* se configura de manera visual. El sistema funciona con una asociación entre palabras clave junto con los métodos de entrada. Posteriormente se relaciona estas palabras clave con las diferentes funciones desarrolladas.

Para asociar una acción o un sistema de ejes, es necesario configurarlo desde los ajustes del proyecto, como se muestra en la ***ilustración 31***. En el caso de este proyecto se han configurado 2 enlaces de acciones, la acción de atacar asociada con un botón, y la acción para salir. Respecto a la asociación de ejes, se han configurado para el movimiento y para la cámara tanto en el eje vertical como el eje horizontal.

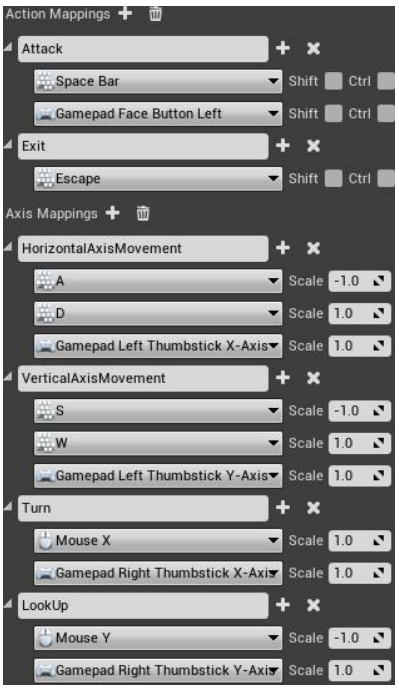

Ilustración 31. Configuración de inputs. Fuente: Elaboración propia.

Este sistema de configuración hace que se agilice el proceso de modificación de métodos de entrada asociados al juego. De esta manera en caso de que se quisiera añadir el soporte para manejo con mando, bastaría con añadir el mapeado de botones y ejes a los ya existentes.

5.3.7. Implementación Hack ‘N’ Slash

Para la implementación del género *Hack ‘n’ Slash* se han creado una serie de elementos que se van a detallar dentro de los siguientes apartados. En primer lugar, se encuentra la implementación del *Pawn* controlado por el jugador. Después, se ha detallado la implementación de los enemigos, incluyendo el sistema sensorial del controlador y el árbol de comportamiento de la *IA*. Posteriormente se explican los elementos desarrollados para el *gameplay*, que son los elementos de generación y curación de daño.

5.3.7.1. Jugador

A la hora de desarrollar el jugador, *Unreal Engine* proporciona una clase llamada *Character* donde ya está contemplado el movimiento y nos da información referente al jugador, como puede ser la velocidad a la que se mueve, aceleraciones, físicas, etc. La implementación en código se ha separado en varios sectores. Por un lado, tenemos el constructor y el método *BeginPlay* que son los encargados de inicializar variables a la hora de empezar la partida. Después tenemos la región de enlazado con los *inputs* del sistema. En estas funciones se enlazan las diferentes entradas con las acciones o los ejes que se han configurado en el sistema.

Por otro lado, tenemos la gestión de daño del personaje. En esta sección se encuentran todos los métodos para generar y recibir daño. Todo el daño que se recibe va a un componente llamado *HealthDamage* creado con ese propósito. Para hacer un sistema de daño donde el jugador pueda hacer combos en los movimientos de ataque encadenando uno detrás de otro ha sido necesario declarar diferentes elementos para poder implementar este aspecto. Además, mediante el uso del gestor de animaciones se hace uso de disparadores o *triggers* que se llaman durante un punto concreto de la animación haciendo que se activen diferentes variables. Una vez están activadas o desactivadas estas características se podrá continuar el combo de ataques o tendrá que empezar de nuevo.

En la ***tabla 3*** se encuentra con más detalle el nombre de los métodos implementados, así como una descripción de su funcionamiento.

Tabla 3. Métodos del actor PlayerPawn

Nombre del método	Descripción
APlayerPawn	Constructor encargado de crear e instanciar los componentes creados.
BeginPlayer	Función de sistema que se ejecuta al crear el elemento en el mapa. Se encarga de inicializar todos los atributos que tiene el jugador.
SetupInputComponent	Encargada de enlazar los controles configurados desde el editor de Unreal Engine con las funciones creadas por código.
HorizontalAxisMovement	Función encargada de recoger el vector necesario para el movimiento en el eje X.
VerticalAxisMovement	Función encargada de recoger el vector necesario para el movimiento en el eje Y.
OnAttack	Disparador del evento creado con la finalidad de asignar animaciones y partículas por Blueprints.
OnMagicChange	Encargada de llamar la actualización del *MagicComponent* del jugador.
ResetCombo_ Implementation	Disparador creado para reiniciar el contador de combo, así como la asignación de animaciones desde Blueprints.
SaveComboAttack_ Implementation	Disparador creado para indicar que el jugador puede continuar el combo de ataques. Modifica el contador de combo, así como la asignación de animaciones desde Blueprints.
SetDamageState	Indica que el jugador puede recibir daño de nuevo.
DamageTimer	Lanza un temporizador para que después de cierto tiempo se reinicie el indicador de recibir daño del jugador.
PlayFlash	Encargada de modificar el indicador para reproducir la animación de daño del jugador.
TakeDamage	Función del sistema encargada de gestionar el daño que recibe el jugador. En nuestro caso, manda el daño recibido al *HealthComponent*.

Nombre del método	Descripción
OnOverlapBegin	Función que se dispara cuando el jugador entra en colisión con un elemento marcado con *Overlap*. Es el encargado de comprobar si está el jugador atacando, si esta colisionando con un enemigo y en caso afirmativo, aplica puntos de daño al enemigo.
OnPlayerDie	Disparador creado para indicar que el jugador ha perdido todos los puntos en el *HealthComponent*. Se encarga de parar todas las animaciones en reproducción y reproducir la animación de muerte del jugador.

5.3.7.2. Enemigos

Para la configuración de la *IA* enemiga es necesaria una serie de elementos que hay que configurar. El jugador ha sido configurado como detectable para la *IA*, será necesario crear un *EnemyController*, que es el encargado de manejar los elementos de los personajes enemigos. Este *EnemyController* es el encargado de comprobar entre todos los elementos registrados en cada uno de los sentidos posibles y comprobar si está detectando el jugador (o cualquier otro elemento que esté configurado como detectable)

Para la gestión de la *IA*, será necesario crear un árbol de toma de decisiones entre ciertas posibilidades. Estos árboles de comportamiento o *Behaviour Trees*, comprueban de entre unas opciones configuradas con una serie de valores si alguna de las condiciones que se evalúan es correcta o incorrecta. El árbol se recorrerá de forma descendente y en sentido de izquierda a derecha en cada actualización de la *IA*, para manejar el comportamiento.

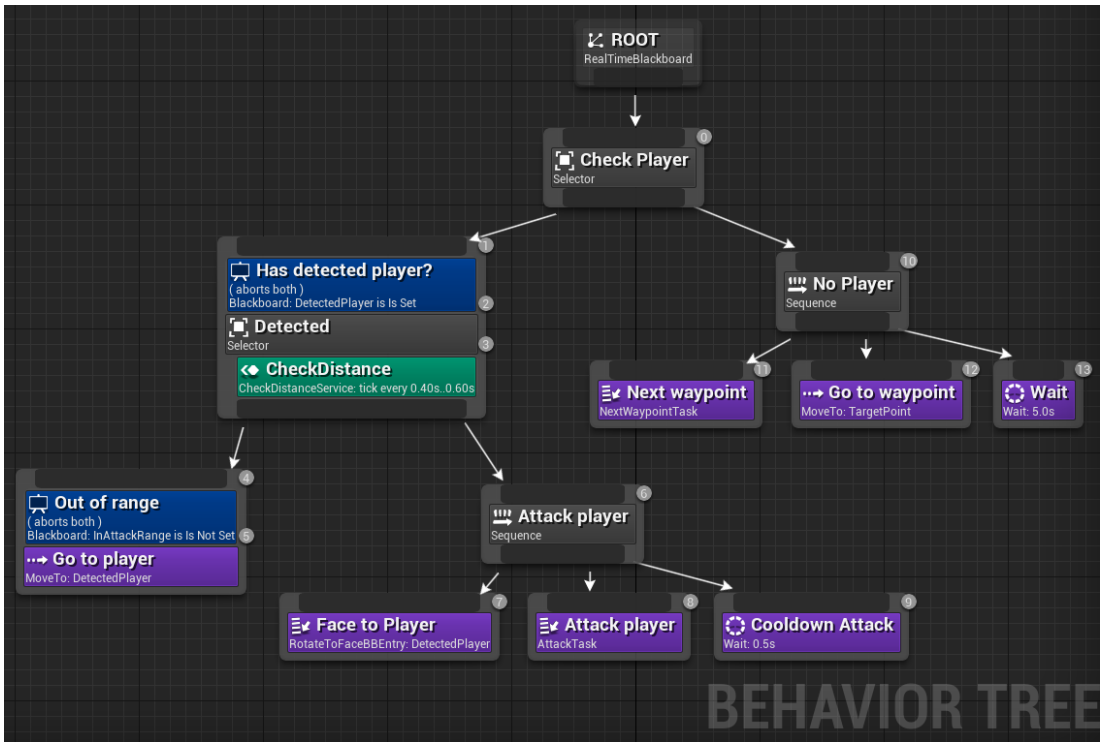

Ilustración 32. Behaviour Tree del enemigo Hack 'n' Slash. Fuente: Elaboración propia

En la ***ilustración 32*** se muestra el árbol utilizado en este proyecto. Como se puede observar no se necesita un árbol muy complejo, ya que lo único que se comprueba es si el jugador está dentro del rango de visión para poder atacarle, y en caso de que no lo esté, el enemigo seguirá haciendo una patrulla por el mapa por los diferentes puntos configurados para ello.

El resultado final de la *IA* enemiga son unos enemigos que son capaces de detectar si el jugador está dentro de su rango de visión, acercarse y atacar en caso de que sea detectado. También es posible que el jugador salga del rango y el enemigo vuelva a su patrullaje corriente, o persiga el enemigo hasta alcanzarlo.

5.3.7.3. Elementos de daño y curación

Para el desarrollo de la partida se han creado dos elementos que se encargan de restarle a la vida del jugador, o sumársela. Debido a la similitud de ambos elementos se va a explicar únicamente el elemento de restauración de vida.

Estos elementos tienen implementado un método encargado de gestionar la colisión y están configurados para interaccionar con el jugador. Estos elementos solo podrán interactuar con actores que tengan un *HealthComponent* incluido. En caso de que no tengan dicho componente, no podrán interactuar con estos elementos.

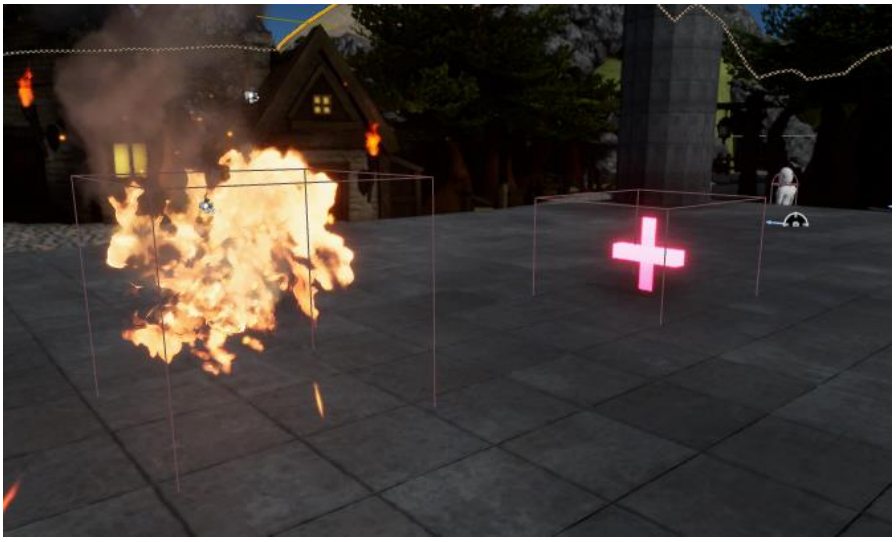

Ilustración 33. Representación gráfica de los elementos de daño y curación. Fuente: Elaboración propia.

En la ***ilustración 33*** se muestran las representaciones graficas de ambos elementos en la escena. En la parte izquierda se representan los elementos que generan daño como fuego y en la parte derecha los elementos de curación con una cruz roja.

5.3.8. Implementación Combate por turnos

La implementación del sistema de combate por turnos se basa en un actor llamado *CombatManager*. Este actor es el encargado de activar el turno de los diferentes actores involucrados en el combate. El jugador interactúa con los personajes involucrados mediante una serie de botones que aparecen en la pantalla y que permiten atacar a los jugadores contrarios. En los siguientes puntos se detallarán los elementos diferenciados de este género.

5.3.8.1. Gestor de combate

Este actor hereda de la clase *Pawn*, ya que no es necesaria la simulación de físicas y demás elementos que implementa la clase *Character*. Para gestionar los diferentes turnos, el gestor de combate posee una matriz con los personajes involucrados en el combate, así como el personaje con el turno actual. Cada personaje tiene asignado con un valor entero el orden de turno.

Para diferenciar la situación en la que haya que mostrar las opciones de ataque se ha creado un indicador que se activa cuando entra en juego algún personaje controlado por el jugador.

Para la notificación entre los elementos que necesiten información del combate, se han creado una serie de eventos que serán activados dependiendo de las condiciones. Estos eventos son para el cambio de turno, notificar cuando finaliza el turno, comprobar el final del combate y conocer si es el turno enemigo. En las ***tablas 4*** y ***5*** se describen los eventos y métodos implementados con más detalle.

Tabla 4. Definición de eventos de CombatManager

Nombre del evento	Descripción
OnChangeTurnEvent	Comprueba si ha terminado el combate, finaliza el turno si es necesario, obtiene el siguiente turno y actualiza la interfaz.
OnOptionSelected	Es llamado cuando se pulsa una de las opciones de atacar y tiene como parámetro el nombre del enemigo objetivo. Se encarga de enviar el mensaje a un jugador para atacar.
OnFinishCombat	Encargado de reproducir la animación de victoria y hacer la llamada para volver al mapa del mundo.
OnTurnEnemy	Encargado de obtener el objetivo del enemigo en este turno.
OnCharacterFinishedTurn	Reproduce una animación para mostrar que se inicia el siguiente turno y llama al evento *OnChangeTurnEvent*.

Tabla 5 Definición de métodos de CombatManager

Nombre del método	Descripción
GetNextCharacter	Encargado de actualizar el indicador de personaje actual, buscar entre todos los personajes involucrados en el combate para obtener el siguiente turno y actualiza la interfaz.
GetNextTurn	Actualiza la lista de participantes y comprueba que haya personajes jugadores vivos. En caso negativo termina el combate.
EndTurn	Método encargado de actualizar el indicador del elemento activo y ocultarlo.
UpdateHUDVisibility	Este método muestra la interfaz del jugador con las opciones y la oculta cuando es el turno del enemigo.
CheckFinishCombat	Comprueba la cantidad de enemigos vivos y la cantidad de jugadores vivos. En caso de que alguno de los dos bandos no le quede ningún personaje vivo, se llamara al evento *OnFinishCombat*.
GetCharacterWithTurn	Obtiene un personaje involucrado en el combate con un turno especifico.
PlayerAttackObjetive	Recibe como parámetro el nombre del jugador objetivo. Se busca entre todos los participantes un jugador con el mismo nombre y dispara el evento *OnAttack* y *UpdateHealth* para el objetivo.
AttackRandomObjetive	Obtiene un jugador aleatorio del bando enemigo de entre todas las opciones viables. Este personaje recibirá el evento *UpdateHealth* como notificación de recepción de daño. El personaje responsable del ataque activará el evento *OnAttack*.

5.3.8.2. Interfaz de usuario

La interfaz presente durante el combate consta de 3 elementos diferenciadores. En primer lugar, están las barras de vida de los enemigos sobre ellos. En segundo lugar, está la zona inferior con las barras de vida de todos los personajes involucrados. Y por último están los botones con las acciones que puede realizar el jugador.

El funcionamiento de la interacción reside en los eventos creados en el gestor de combate. La interfaz tiene acceso a los eventos creados en el gestor, de forma que, en el momento que se pulse un botón con una acción, el gestor de eventos recibe una notificación con la opción seleccionada. De esta manera el gestor de combate notifica a los jugadores que han atacado o han sido atacados.

5.3.9. Sigilo

La implementación de este género tiene varios elementos diferenciadores del resto. El sistema se basa en evitar ser descubierto por los enemigos. Para estar oculto y ser descubierto se han creado unas luces que se encargaran de activar la detección del enemigo. En caso de que el jugador sea atrapado por un enemigo, se reiniciará el nivel y tendrá que empezar de nuevo.

Estas luces contienen una caja de colisión que se encarga de activar todo el sistema de detección de los enemigos. Respecto a la implementación del jugador, es similar a la implementada con los actores del género *Hack 'n' Slash*, pudiendo mover la cámara y moverse en todas direcciones.

La implementación del enemigo contiene un *AIController* para el manejo del *Pawn*, así como un *Behavior tree*. La implementación de estos enemigos es la misma que la planteada con los enemigos del *Hack 'n' Slash* a excepción de que a estos enemigos no se les ha creado un sistema de patrulla.

Por otro lado, se ha creado el objetivo del nivel. Este objetivo se activará cuando el jugador entre en contacto con él, activando un evento para terminar el nivel. Una vez se termine el nivel, se mostrará el mensaje de victoria y se mostrará el menú principal.

6. Conclusiones

En este apartado se extraen las conclusiones del trabajo realizado en este proyecto a partir del grado de consecución de los objetivos planteados al inicio de este mismo. El objetivo principal de desarrollar un videojuego multigénero en *Unreal Engine 4* se ha subdividido en objetivos más concretos, descritos en el apartado ***3. Objetivos***, y hay que decir que todos ellos han sido conseguidos en su totalidad. El primer paso para conseguir fue centrarse en el aprendizaje del desarrollo en *Blueprints* y *C++*. Una vez obtenida una base sólida de conocimiento en ambos sistemas, se planteó la combinación de ambas opciones de desarrollo, objetivo que se completó también satisfactoriamente. El desarrollo de un género concreto puede abarcar desde los aspectos y mecánicas más simples para definir el género hasta el desarrollo complejo de un estudio con un amplio equipo de desarrolladores. Por este motivo, para la resolución de la mayoría de los problemas que se han ido encontrando durante el desarrollo, ha sido necesario visualizar, modificar, crear y adaptar técnicas y elementos a medida que se iban implementando. Una vez se coge soltura con el entorno de *Unreal Engine* y aumenta la velocidad de desarrollo es muy práctica la implementación de elementos con el sistema de *Blueprints*.

Durante la primera parte del desarrollo, a la hora del aprendizaje cabe mencionar que la mayor parte de cursos y tutoriales que he encontrado han sido en el sistema *Blueprints*, aunque una vez familiarizado con este sistema se entiende mejor y se puede avanzar más rápido. Para desarrollar en *C++* es necesario un equipo con mayor potencia debido a que las compilaciones que hace el sistema de *Unreal* consumen muchos recursos y tiempo, sin embargo, con el sistema *Blueprints* no es necesario disponer de un equipo con prestaciones altas ya que la compilación apenas dura unos segundos. En beneficio del desarrollo en *C++* he de decir que la traducción de varias líneas de instrucciones en *C++* al sistema *Blueprints* puede ser complicada ya que conlleva establecer una maraña de conexiones. También hay que remarcar que con el sistema de conexiones de los *Blueprints* hay que tener especial cuidado en el orden de dichas conexiones ya que, al ser un sistema totalmente visual, es muy probable perder el hilo del desarrollo y cometer errores.

Respecto al acabado del proyecto, estoy muy contento con el resultado final, ya que he conseguido realizar un videojuego completo con tres géneros diferenciados por niveles, aunque hay aspectos a los que me habría gustado dedicarles más tiempo, como por ejemplo a las técnicas de iluminación, la creación de mundos procedurales, la exportación a

dispositivos móviles, etc. Para terminar, este trabajo supone el comienzo de una carrera dentro del sector del desarrollo de los videojuegos, un mundo que me apasiona desde que elegí cursar esta titulación y al que quiero dedicarme profesionalmente a partir de ahora.

6.1. Control de tiempos en Toggl

En este apartado se va a hacer un análisis del registro de horas realizado con la herramienta *Toggl*. Como se puede observar en la ***ilustración 34,*** el tiempo dedicado al proyecto es ligeramente superior al estimado respecto de la conversión de créditos, que serían 300 horas. Respecto a la programación se ha dividido en 4 categorías, en primer lugar, esta *Unreal Engine*, que engloba todo el desarrollo en la plataforma. En segundo lugar, está la documentación englobando todo lo relacionado con la redacción e investigación de este documento. En tercer lugar, está el aprendizaje de *Unreal Engine* donde se recopilan todas las horas invertidas en seguimiento de cursos y tutoriales para familiarizarse con el entorno. Por última parte, está la categoría de modelado y animación, donde se han almacenado todos los elementos relacionados con esta parte del proyecto.

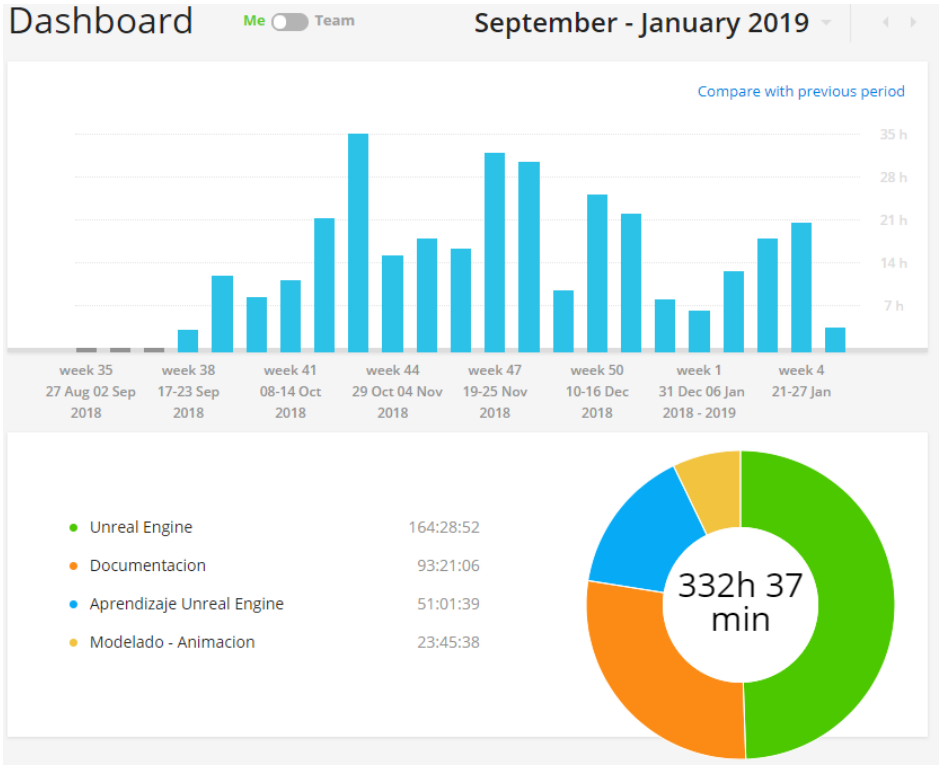

Ilustración 34. Resumen de tiempos en Toggl

7. Bibliografía y referencias

1. La industria del videojuego genera más de 8.700 empleos en España. Recogido en diciembre 2018 de Diario Marca. https://www.marca.com/tiramillas/videojuegos/2018/11/27/5bfd3b01e2704edf3c8b4678.html
2. Industria del videojuego. Recogido en noviembre 2018 de AEVI (Asociación española de videojuegos). http://www.aevi.org.es/la-industria-del-videojuego/en-el-mundo/
3. Desarrollo de videojuegos independientes (*indie*). Recogido en enero 2019 de Wikipedia. https://es.wikipedia.org/wiki/Desarrollo_de_videojuegos_independiente
4. Videojuego Triple A. Recogido en diciembre 2018 de *Wikipedia*. https://es.wikipedia.org/wiki/AAA_(industria_del_videojuego)
5. Definición de videojuego. Recogido en enero 2019 de RAE (Real Academia Española). http://dle.rae.es/?id=bmnbNU7
6. *PEGI. (Pan European Game Information).* Recogido en enero 2019 de *Pegi.info*. https://pegi.info/es
7. Género *Hack 'n' Slash*. Recogido en diciembre 2018 de *Wikipedia*. https://es.wikipedia.org/wiki/Hack_and_slash
8. Género *RPG*. Recogido en diciembre 2018 de *Wikipedia*. https://es.wikipedia.org/wiki/Videojuego_de_rol
9. Pokémon entra en el *Guinness* de los récords. Recogido en noviembre 2018 de *IGN*: https://www.ign.com/articles/2009/01/17/pokemon-report-world-records-edition
10. Referencia *Unreal Engine*. Recogido en diciembre 2018 de *Unreal Engine*. https://www.unrealengine.com/en-US/what-is-unreal-engine-4
11. *Cel Shading*. Recogido en noviembre 2018 de *Wikipedia*. https://es.wikipedia.org/wiki/Cel_shading
12. Especificación de propiedades en *Unreal Engine*. Recogido en enero 2019 de *Unreal Engine*. https://docs.unrealengine.com/en-us/Programming/UnrealArchitecture/Reference/Properties/Specifiers

8. Anexo: Arte

8.1. Jugador géneros Hack ‘n’ Slash y combate por turnos

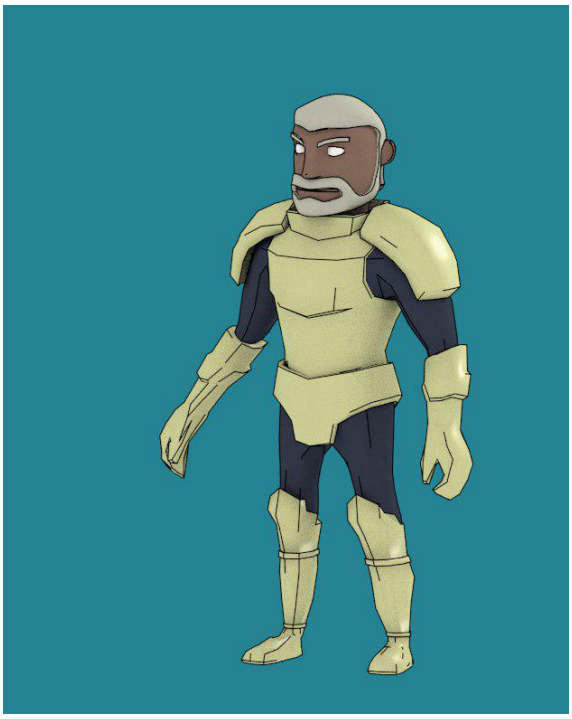

Ilustración 35. Renderizado del personaje desde Blender.

Ilustración 36. Personaje integrado en Unreal Engine

8.2. Sigilo

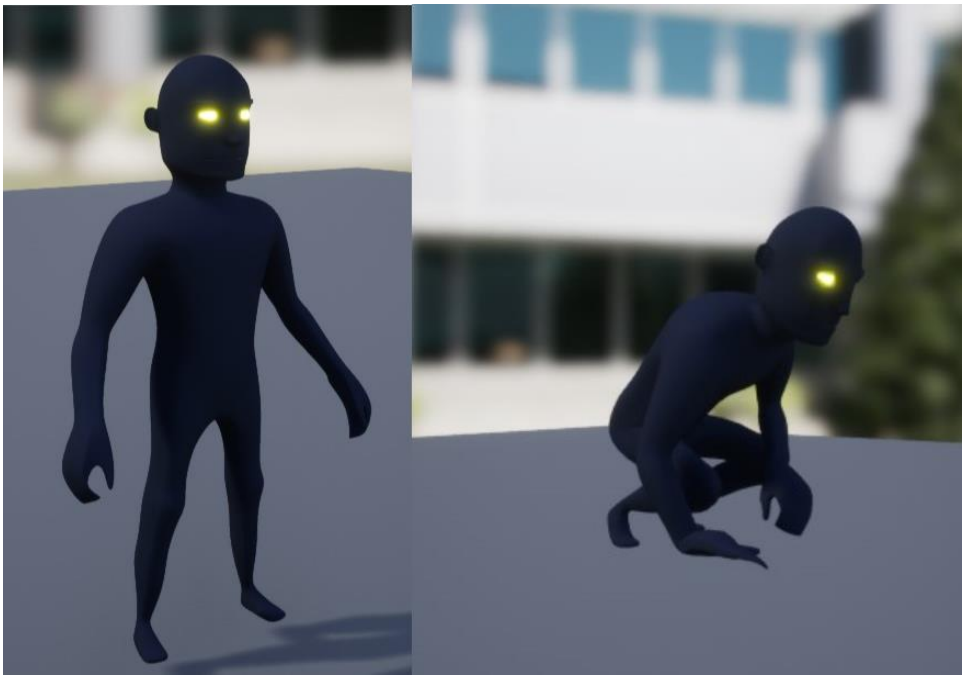

Ilustración 37. Jugador Sigilo integrado en Unreal Engine.

Ilustración 38. Enemigo genero sigilo integrado en Unreal Engine

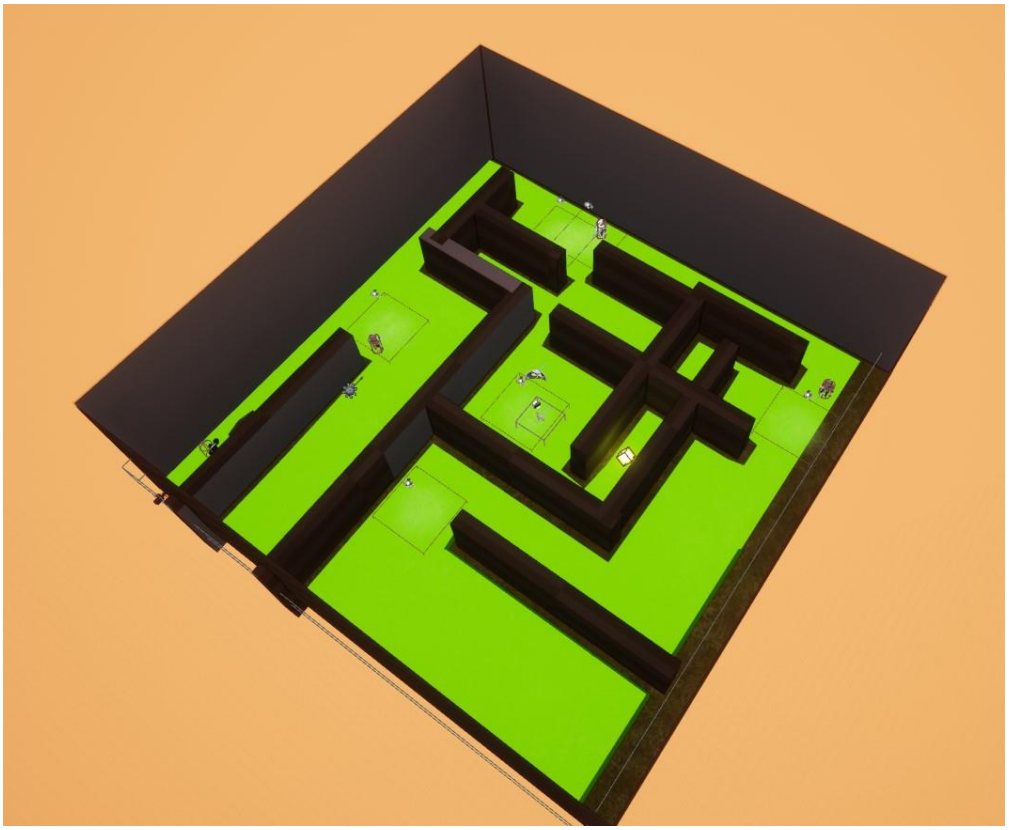

Ilustración 39. Vista aérea del nivel en género sigilo desde editor.

Ilustración 40. Captura género sigilo ingame 1.

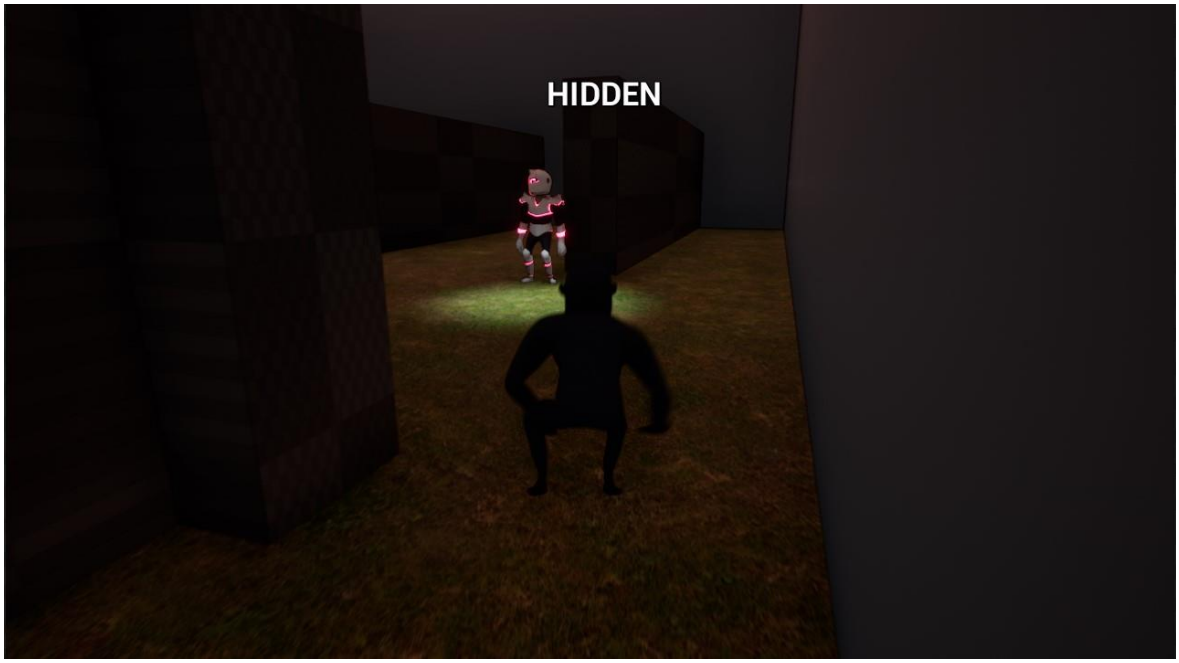

Ilustración 41. Captura género sigilo ingame 2.

Ilustración 42. Captura género sigilo ingame 3.

Ilustración 43. Captura género sigilo ingame 4.

Ilustración 44. Captura género sigilo ingame 5.

8.3. Combate por turnos

Ilustración 45. Enemigo género RPG con combate por turnos integrado en Unreal Engine.

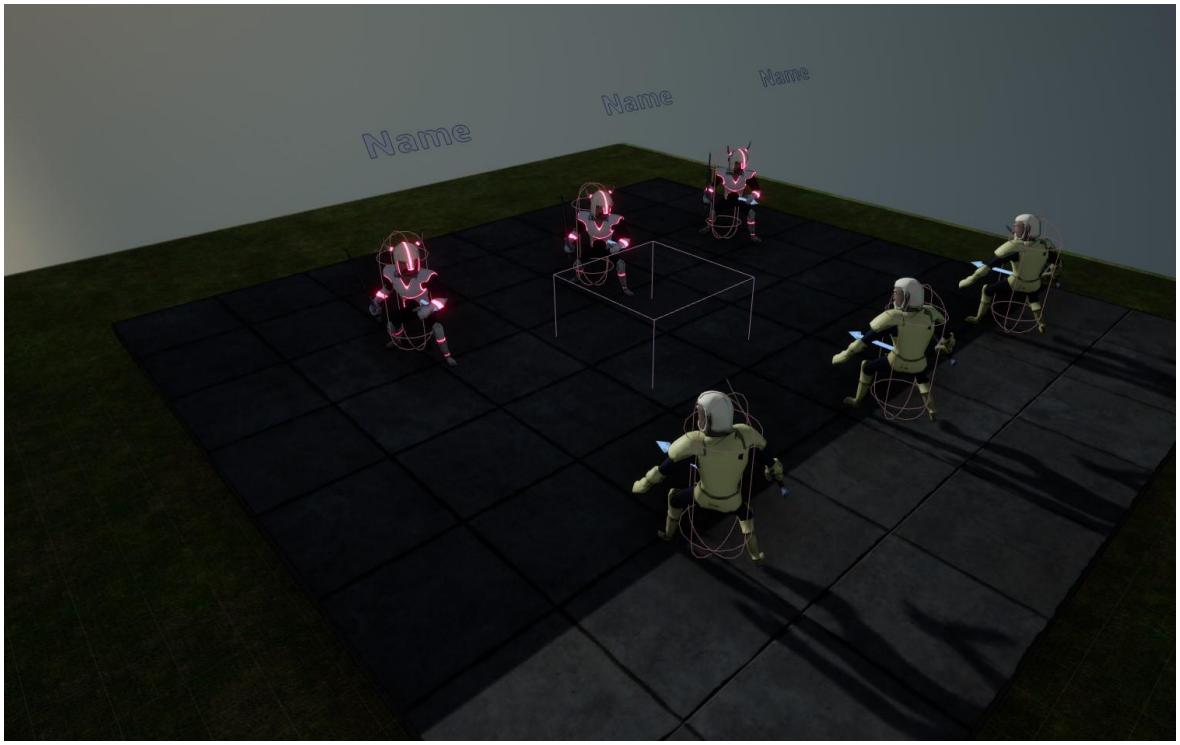

Ilustración 46. Mapa del género RPG con combate por turnos en el editor de Unreal Engine.

Ilustración 47. Captura género RPG con combate por turnos ingame 1.

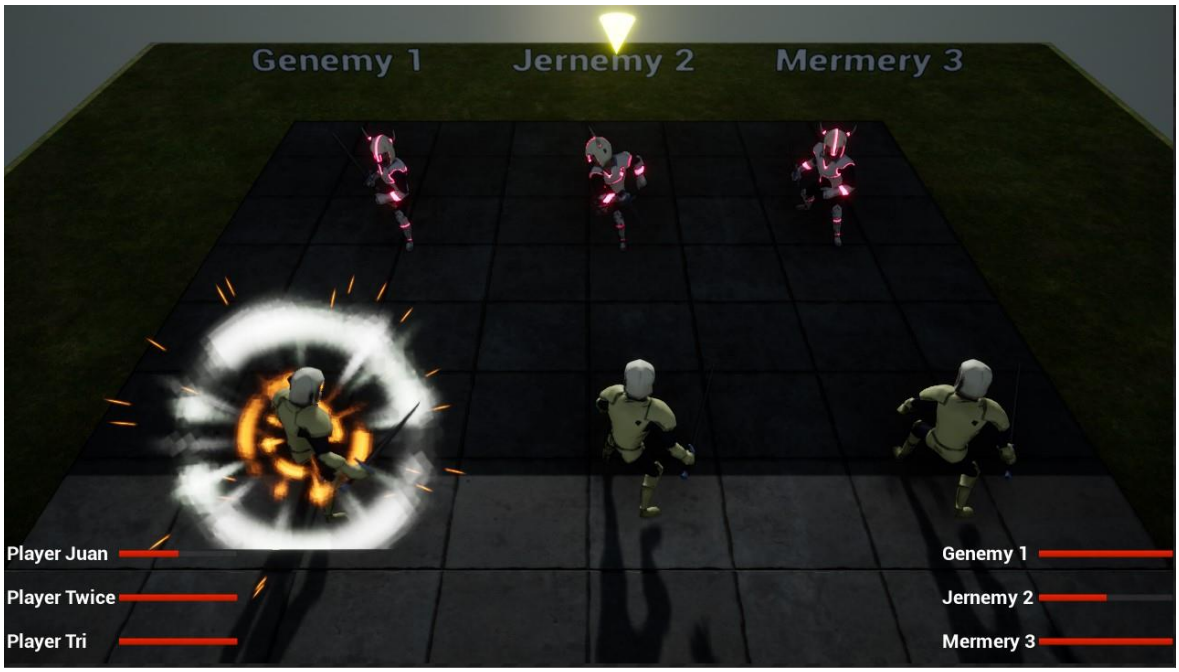

Ilustración 48. Captura género RPG con combate por turnos ingame 2.

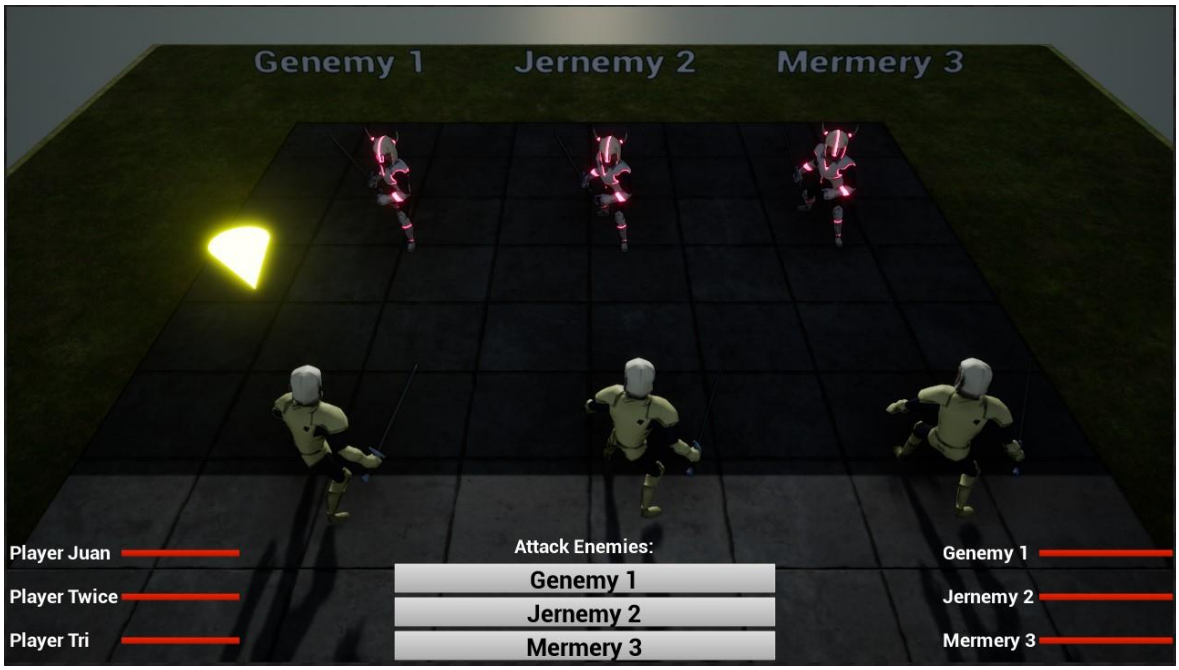

Ilustración 49. Captura género RPG con combate por turnos ingame 3.

Ilustración 50. Captura género RPG con combate por turnos ingame 4.

8.4. Hack ‘n’ Slash

Ilustración 51. Enemigo género Hack 'n' Slash integrado en Unreal Engine.

Ilustración 52. Captura género Hack 'n' Slash ingame 1.

Ilustración 53. Captura género RPG con combate por turnos ingame 2.

Ilustración 54. Captura género RPG con combate por turnos ingame 3.

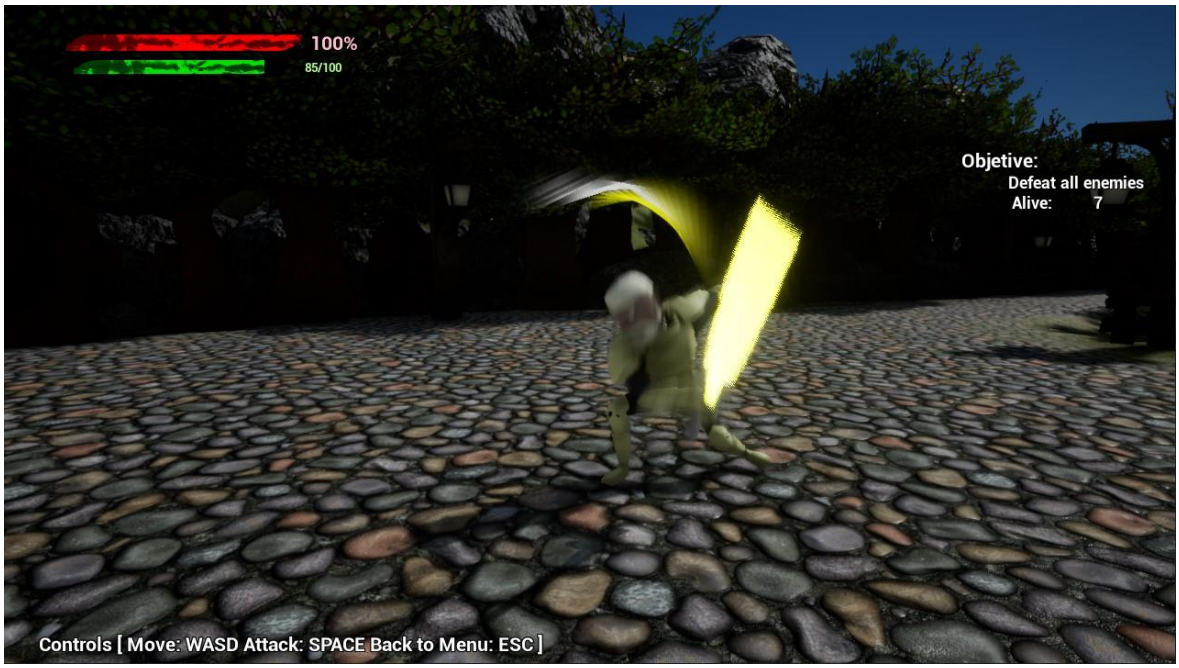

Ilustración 55. Captura género RPG con combate por turnos ingame 4.

Ilustración 56. Captura género RPG con combate por turnos ingame 5.

Ilustración 57. Captura género RPG con combate por turnos ingame 6.

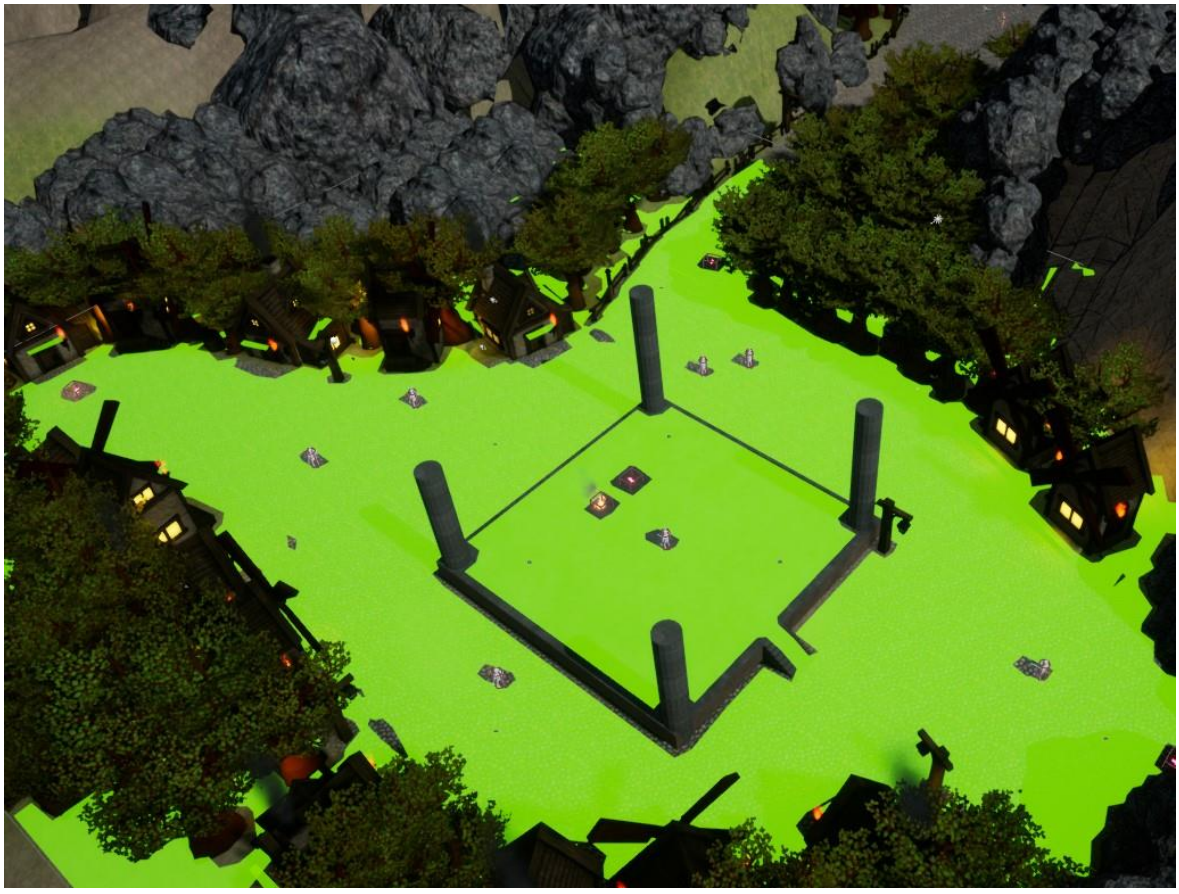

Ilustración 58. Vista aérea del mapa género Hack 'n' Slash.